WL-TR-95-1118

AVIONICS SOFTWARE REENGINEERING
TECHNOLOGY (ASRET) PROJECT
REENGINEERING TOOL (RET)
USER'S MANUAL

D.E. WILKENING
J.P. LOYALL

TASC
55 Walkers Brook Drive
Reading, Massachusetts 01867

MAY 1995
Project Final Report for 5/1/92 – 5/1/95

19960325 115

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-7409
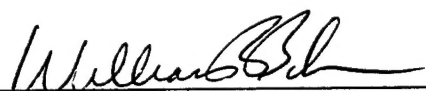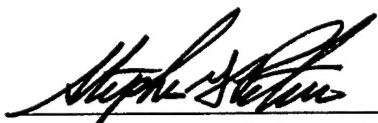
DTIC QUALITY INSPECTED 5

NOTICE

When Government drawings, specifications or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

KENNETH LITTLEJOHN, Project Engineer
Software Concepts Section
WL/AAAF-2

WILLIAM R. BAKER, Acting Chief
Avionics Logistics Branch
WL/AAAF

STEPHEN G. PETERS, Lt Col, USAF
Deputy Chief
System Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify WL/AAAF, WPAFB, OH 45433-7301 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE May 1995	3. REPORT TYPE AND DATES COVERED May 92 to May 95

4. TITLE AND SUBTITLE
Avionics Software Reengineering Technology (ASRET)
Tool (RET) User's Manual

6. AUTHOR(S)
D.E. Wilkening , J.P. Loyall (TASC)

5. FUNDING NUMBERS
C F33615-92-D-1052
PE   78012
PR   3090
TA   01
WU   14

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
TASC, Inc.
55 Walkers Brook Drive
Reading, MA 01867

8. PERFORMING ORGANIZATION REPORT NUMBER
TASC: TR-06661-6

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Avionics Directorate
Wright Laboratory
Air Force Material Command
Wright Patterson AFB OH 45433-7409

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-95-1118

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report explains how to use the Reengineering Tool (RET) prototype developed under the Avionics Software Reengineering Technology (ASRET) project. The RET prototype is a software reengineering tool that assists in improving and translating avionics simulation software written in FORTRAN to Ada.

14. SUBJECT TERMS
Reengineering, Reverse Engineering, Reuse

15. NUMBER OF PAGES
62

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

STANDARD FORM 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

MI009
1-22-92

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.   INTRODUCTION

This report explains how to use the Reengineering Tool (RET) prototype developed under the Avionics Software Reengineering Technology (ASRET) project. The RET prototype is a software reengineering tool that assists in improving and translating avionics simulation software written in FORTRAN to Ada.

1.1   BACKGROUND

1.1.1   Context

We conducted the ASRET project under the Avionics Software Technology Support (ASTS) program. The ASTS program is an ongoing activity of the Software Concepts Group, Avionics Logistics Branch at Wright Laboratory (WL/AAAF-3), Wright Patterson Air Force Base in Ohio. The objective of the ASTS program is to perform research and development for enhancing Embedded Computer System (ECS) software development and postdeployment support.

The main objective of ASRET was to develop an automated Reengineering Tool (RET) prototype for avionics support software. The specific goals included investigating existing reengineering and reverse engineering processes, techniques, and software tools, defining a reengineering process model, and building the RET prototype software tool.

1.1.2   Rationale

The reengineering of software from one language to another is becoming a necessity as Air Force organizations strive to modernize and improve the maintainability of their systems while avoiding the excessive costs of new development. Systems that have been in use for years often incur large maintenance costs for a number of reasons.

- Continual maintenance has made the current implementation and original design inconsistent, the code harder to understand and error-prone, and the documentation out-of-date.

- They are written in languages that have fallen out of favor. The limited selection of support tools for these languages, the corresponding expense of the associated support tools, and the shrinking pool of qualified programmers to maintain the software adds to the expense of maintenance.

- They were developed without modern software engineering practices, resulting in code that lacks structure and is difficult to understand.
- Employee turnover has reduced the staff's understanding and intimate knowledge of the systems.

Wright Laboratory initiated the ASRET project to help reduce maintenance costs for legacy systems and to assist in the evolution to Ada. To this end, we developed an environment for reengineering software from one language to another. We concentrated on the reengineering of avionics simulation software written in FORTRAN to Ada, and designed the RET prototype so that additional languages could be supported in the future.

1.1.3 Goals

The objective of the ASRET project was to develop an automated RET prototype for avionics support software. The specific goals included investigating existing reengineering and reverse engineering processes, techniques, and software tools, defining a reengineering process model, and building a RET prototype that supports:

- Translating avionics simulation software written in FORTRAN to Ada
- Improving the software through restructuring techniques
- Changing the design of the software so that it is consistent with modern software engineering principles
- Adding documentation that is consistent with the software.

1.2 REPORT ORGANIZATION

Section 1 introduces the ASRET project and highlights the rationale for, and goals of the Reengineering Tool (RET). Section 2 provides an overview of the RET approach to reengineering and describes the views available through the RET user interface. Section 3 presents the ASRET Reengineering Process Model. Section 4 illustrates how to apply the process model and demonstrates, in the context of reengineering a sample application, techniques that the RET supports. Section 5 is a reference for the RET User Interface. Appendix A defines acronyms used in this document.

The *Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume I, Project Summary, Account, and Results* (Ref. 1) relates the lessons that we learned while developing the RET prototype, and provides information that may help you fully exploit the capabilities of the RET.

The *Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume II, Reengineering Tool (RET) Diagrams* (Ref. 2), recounts our experiences in exercising the RET prototype and provides annotated output of the RET. This information may help you interpret diagrams that you create with the RET prototype.

2.   RET OVERVIEW

2.1   THE GENERAL APPROACH

The RET comprises two distinct logical parts called the Left-Hand Side (LHS) and the Right-Hand Side (RHS). The LHS provides views of the original FORTRAN program, or subject system. The RHS provides views of the Ada program being developed (i.e., the target system). The LHS allows you to navigate and view aspects of the subject system, but does not support changing the subject system.

The RHS supports constructing, refining, viewing, and navigating the target system. You may construct a basic structure for the RHS (macro restructuring) using information extracted from the LHS. Once the basic structure of the RHS is established, you may refine the target system (micro restructuring) on the RHS.

Semiautomated RET components support construction activities; they suggest large-scale reorganizations of the subject system and populate the RHS with the basic structure of the target system. The components that support refinement allow you to intervene to apply your knowledge of the subject system and application domain and insight, which is beyond the semiautomated support provided by the RET, to modify and improve the RHS representations.

The RET prototype supports engineering an Ada program by reusing and transforming parts of the FORTRAN program. The process is iterative as illustrated in Figure 1.

- You may explore views of the original FORTRAN program that the RET generates on the LHS.
- You may select LHS entities such as subroutines, statements, or data elements of the original FORTRAN program.
- The RET transforms the LHS entities and incorporates them into the RHS.
- You may explore views representing the Ada program on the RHS.
- You may interactively or automatically refine the RHS through the views.

You may repeat the cycle, exploring the LHS to select additional FORTRAN entities to reuse in building the RHS. The graphical user interface presents the LHS and RHS views while the object-based database manages the underlying data structures or internal representations.
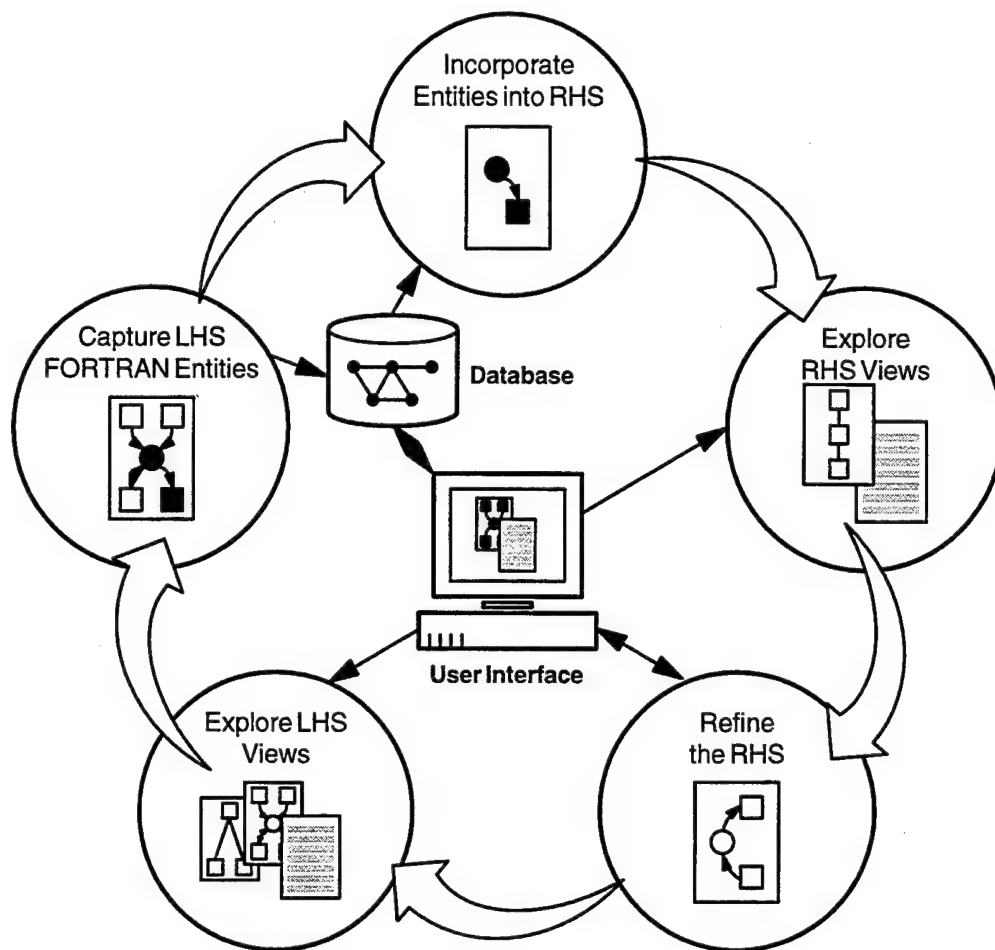
Figure 1   Developing Ada by Reusing FORTRAN

The RET approach to reengineering is to create a new program on the RHS, reusing components of the LHS. Structured design and programming principles are compatible with the RET prototype and the ASRET Process Model described in Section 3.

The specific steps that you may take to apply the ASRET Process Model when using the RET prototype are illustrated in Figure 2.

1.   The RET prototype constructs the package and subprogram structure for the RHS. It captures the subprogram structure from the LHS, transfers it to the RHS, and clusters the subprograms into Ada packages.

2.   You may refine the RHS structure so that related subprograms and data items are grouped together into packages.

3.   The RET prototype moves data items and type declarations from the LHS to the packages and subprograms on the RHS to which they are most closely related.

5
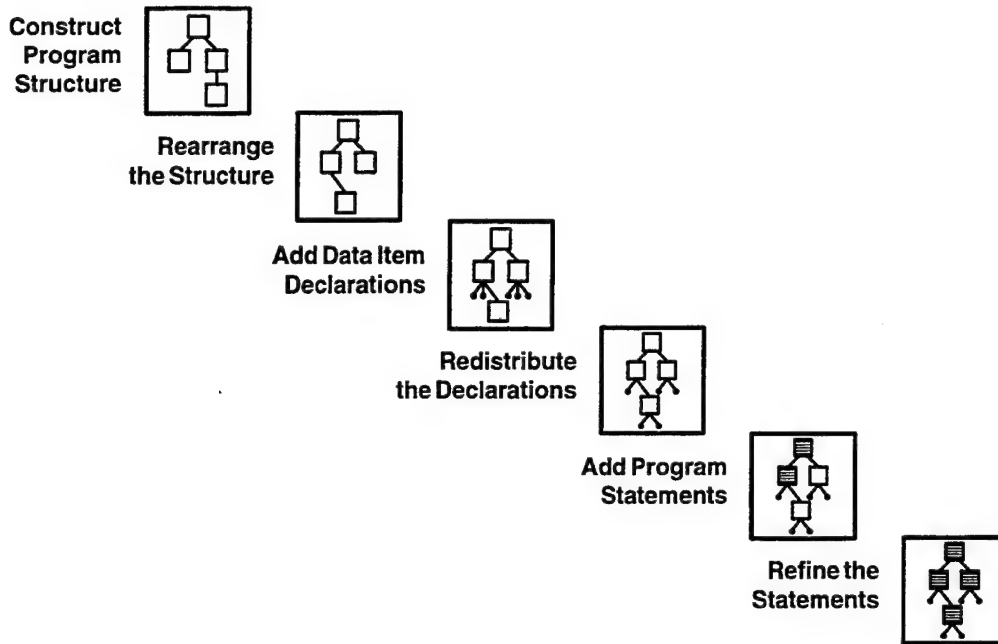
Figure 2 Incorporating Macro and Micro Entities

4. You may then refine and redistribute the declarations. For example, data items that are closely related but used by several subprograms might be in different packages, and thus need to be grouped together. As another example, data items might be moved into or out of a package's private part to reflect their scope.

5. At this point, the RHS contains the modular structure of the program. The RET prototype then transforms statements from the LHS and moves them to the bodies of the RHS subprograms and packages.

6. You may then refine individual statements on the RHS to tune the RHS structure.

2.2 THE USER INTERFACE

The RET provides the views listed in Table 1.

Table 1 Reengineering Tool (RET) Views

LHS	RHS	VIEW	NAME	DISPLAYS
✓	✓	SCL	Source Code Listing	FORTRAN or Ada source code
	✓	PACK	Packager Diagram	Ada package structure
✓		DED	Declaration Diagram	FORTRAN declaration nesting structure
✓		CD	Call Diagram	FORTRAN subprogram calling structure
	✓	DFD	Data Flow Diagram	Data flow through the Ada program

6

- The Source Code Listing (SCL) shows the FORTRAN source code after processing by the SPAG component of plusFORT (Ref. 3) on the LHS and the generated Ada code on the RHS.

- The Packager view (PACK) is a graphical view that shows the package and subprogram nesting structure, and provides an interface for developing the Ada package structure on the RHS.

- The Declaration Diagram (DED) is a textual view that documents the FORTRAN system declaration structure. The DED provides information on common blocks, subprograms, and data objects.

- The Call Diagram (CD) is a textual view that documents the FORTRAN calling structure. The CD shows which subprograms call, or are called by other subprograms.

- The Dataflow Diagram (DFD) is a graphical view that documents the Ada system data flow. The DFD is a hierarchy of dataflow graphs that show, in increasing detail, how data repositories are transformed by the Ada subprograms.

2.2.1   SCL

The Source Code Listing (SCL) is a textual view that shows the FORTRAN or generated Ada source code. The LHS SCL displays the FORTRAN code as it was formatted during input to the RET prototype, after it was processed by SPAG. SPAG alters the source code, so the output format is generally different from that of the original unprocessed FORTRAN code. The SCL shows individual FORTRAN subprograms.

2.2.2   PACK

The Packager view (PACK) is a hierarchy of graphs. The hierarchy corresponds to the nesting structure of the target system. There is one graph for the Ada library, and one for each potential package. The nodes in each graph represent modules, i.e., subprograms or packages, and the edges represent either data binding relationships or subprogram calls. Section 4.1.1 describes the Packager view and how it is used to cluster a sample application.

The Packager view contains two kinds of edges. Thin, undirected edges depict the data sharing relationships between two nodes. A thin edge between two subprograms indicates that the two subprograms share data. A thin edge between a package, P, and a package or subprogram node indicates that at least one subprogram in P shares data with the modules that the other node represents. The edge labels list the data objects or show the subprograms in the package that share data.

7

A thick edge drawn as an arrow directed from one subprogram to another represents a subprogram call in the direction of the arrow. If the edge is drawn between a package and another node, then the edge may represent multiple subprogram calls and its label shows either the number of subprogram calls or the names of the called subprograms.

2.2.3 DED

The Declaration Diagram (DED) is a textual view that shows the declaration structure of the subject system. The DED lists the subprograms, common blocks, and data objects for the FORTRAN system. For each subprogram, it lists the formal parameters, local constants and variables, and included files. For each variable, constant, and parameter, the DED shows the data type and describes where the data object is declared and referenced.

2.2.4 CD

The Call Diagram (CD) is a textual view that shows the calling structure of the FORTRAN system. The CD lists the subprograms that comprise the original system and shows the subprograms that each one calls, and the subprograms that are called by each one. The CD initially shows a list of subprograms and allows you to view the additional calling information for the subprograms of your choice.

2.2.5 DFD

The Dataflow Diagram (DFD) view is a hierarchy of dataflow graphs. The hierarchy corresponds to the calling structure of the target system. There is one graph for each subprogram *declared in the target system* that is called. This means that there are no graphs associated with undefined external subprograms or intrinsic functions (because you may not have their source code).

The DFD contains **transform** and **repository nodes**. Three kinds of transform nodes model programs:

1. **Nonterminal (or call) nodes** represent subprograms that are associated with graphs because they call other subprograms.
2. **Terminal (or leaf) nodes** model subprograms that don't call any other subprograms and thus have no graphs.
3. **Body nodes** represent subprogram bodies.

Two flavors of repository nodes model data.

1. **Buffer nodes** represent data objects (local variables, global variables such as those declared in Ada packages, and subprogram parameters).
2. **Repository collection (or record) nodes** combine sets of repository nodes.

8

Arrows between the transform and repository nodes indicate the flow of data. Transform nodes are labeled with subprogram names. Repository nodes are labeled with data item names. Repository collection nodes representing more than a few (you may specify the threshold) data items display the total number of data items, and you may click on the collection nodes to view the individual data item names. The collections may be nested.

The DFD doesn't show nodes for intrinsic functions or external subprograms. Intrinsic functions are those listed in Appendix D.3 of the VAX FORTRAN Language Reference Manual (LRM) (Ref. 4). Examples are SIN and SQRT. External subprograms are system routines, listed in Appendix D.4 of the LRM, that are called from the FORTRAN system. Examples are DATE and EXIT.

The DFD automatically eliminates redundant transform nodes. For example, if module A called module B three times and different actual parameters are supplied for each call, then the diagram for module A would contain three distinct nodes, each labeled "B," representing the three calls to module B. The three nodes would only be connected to different repository nodes in the diagram if different actual parameters were used in each of the calls. The RET prototype only generates one "B" node if the same actual parameters are used in each call.

The DFD may combine sets of repository nodes into collections. This is analogous to combining several variables into a record structure. The strategy reduces the number of repository nodes in a graph. It is most effective if each of the repository nodes in a collection are referenced more or less by the same set of modules. The capability is experimental so the RET prototype doesn't provide a user interface for specifying the groupings. You may specify the grouping by following the example in the RET prototype source code file "dfd-record-map.re" to produce collections.

3. REENGINEERING PROCESS MODEL

The software reengineering process model for the RET prototype is illustrated in Figure 3. Steps in the process label the boxes in the figure, and inputs and outputs for each step label the icons between boxes.
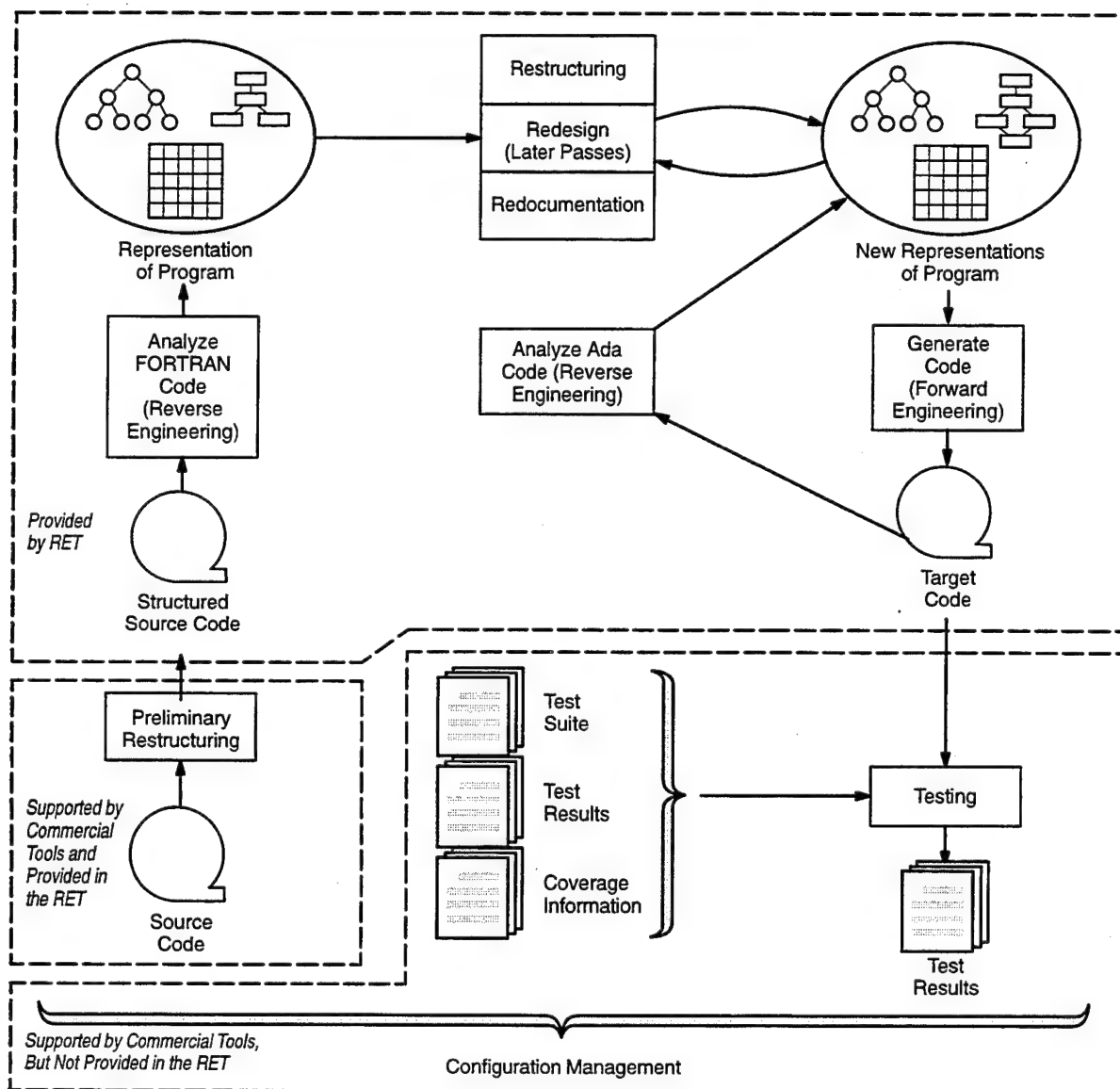
Figure 3   ASRET Reengineering Process Model

10

The process model specifies a set of tasks (the steps of the process) that should be performed and the sequence in which they should be performed to reengineer a program written in FORTRAN to Ada. The process model also specifies the information necessary and desirable to support these tasks. The process model does not specify how the tasks are to be performed (i.e., they might be automated, as many are in the RET, or they might be performed manually).

The first step in the process model is to perform some preliminary restructuring of the source code of the original implementation. Preliminary restructuring improves the layout of the source code by removing unstructured program constructs, such as GOTO statements, dead code, and implicit types. Preliminary restructuring is separated from the later restructuring step because it can be completely automated by commercial tools, and placed first in the process model because the structured version of the source program is usually easier to analyze, understand, and restructure.

After preliminary restructuring is complete, the RET analyzes the improved source code and constructs representations of the program. Some of the representations, such as abstract syntax graphs (ASGs) and symbol tables, are machine-readable representations used only by automated restructuring and redesign tasks. Others, such as flow graphs and structure charts, aid in program understanding, redocumentation, and manual restructuring. For manual restructuring, the set of representations contains a source code listing.

You may perform the restructuring, redesign, and redocumentation steps multiple times, each time building upon the results of the previous pass. A multipass approach is necessary because it is easier and less error-prone to reengineer a large program in stages, verifying the program after each pass. Restructuring (i.e., changing the structure of the program without changing its functionality) is performed first, possibly in several passes. These passes perform the following steps:

1. *Macro control restructuring* groups statements and control structures of the program into modules, such as procedures, functions, and packages. This includes recovering modules of the original program, generating new modules, and specifying a declaration nesting structure for modules.

2. *Macro data restructuring* groups data items, such as types, variables, and constants, and associates them with modules created during macro control restructuring. This includes recovering data groupings of the original program, creating new groupings, and creating abstract data types and records.

3.   *Micro control restructuring* manipulates individual control structures. This includes the translation of individual statements and functionality-maintaining alterations, such as code lifting (Ref. 5).

4.   *Micro data restructuring* manipulates individual data items. This includes actions such as translating, changing names, changing types, creating symbolic constants, and changing the scope of variables.

Macro control and data restructuring should be performed first to develop a modular structure for the target system, followed by micro control and data restructuring to restructure individual components of the program.

After restructuring is complete, the RET prototype generates code in the target language and the program is tested to ensure that the restructuring did not introduce any errors or undesired functional changes. The test data of the original program can be used and the results compared with the results of testing the original program. In many cases, the test data will need to be reengineered to work with the reengineered program.

Any differences in the results of testing indicate the introduction of an unexpected functional change during restructuring. Coverage analysis must be performed during the testing of the target code because restructuring can introduce or alter control and data characteristics of the program. When an error in the target program is indicated, the program can be corrected by amending the target code directly or by restructuring the representations and regenerating the target code.

Once you have restructured the program and created a functionally equivalent program in the target language, you may perform additional restructuring and redesign actions on the program. These steps use the same set of actions (i.e., macro control, macro data, micro control, and micro data), but have different goals.

Further restructuring improves the structure of the program without changing its functionality. The goal of redesign is to change the functionality of the program (e.g., to correct design flaws or improve the design). If you edited the target program code to correct errors indicated during testing, the RET analyzes the code to generate representations before performing subsequent restructuring and redesign. The RET performs redocumentation simultaneously with the restructuring and redesign steps and can save the generated representations for documenting the program structure and design.

The RET reengineering process model includes modern software development processes, such as continuous testing, iterative restructuring and redesign, and configuration management. The process model is a specialization of the Chikofsky-Cross process model (Refs. 6, 7). The entire Chikofsky-Cross model is represented, although there are differences:

- Program management extensions to the process model (Ref. 8) are included, such as configuration management and testing.

- Easily automated steps, such as preliminary restructuring, are separated so they can be addressed by commercial tools.

- Chikofsky-Cross steps are decomposed, such as restructuring into macro control, macro data, micro control, and micro data restructuring.

- Iteration steps that are implicit in the Chikofsky-Cross process are explicitly introduced.

4.   APPLICATION OF CONCEPTS

The RET helps you develop an Ada system by reusing parts of the existing system. It supports, but does not enforce, the ASRET process model by implementing macro and micro restructuring as defined in Section 3. You may first apply macro restructuring features to construct a skeleton of the Ada system. The skeleton provides the modular structure and the distribution of variables among the modules. You may then explore and refine the Ada structure, and add program statements using micro restructuring features of the RET.

This section describes some problems that you may face when reengineering a legacy system, i.e., a system that has undergone many modifications through years of maintenance, and explains how the RET helps you overcome those problems. Section 4.1 describes the use of the *Packager* component of the RET. Section 4.2 discusses the use of the *Transformer* component and explains some of the issues involved in translating certain language features.

4.1   DEVELOPING PROGRAM STRUCTURE USING THE PACKAGER

You may peruse the FORTRAN system by navigating through several views. You may find yourself overwhelmed with information when you initially confront an entire legacy system if it is very large. One way to reduce the amount of information that you must comprehend is to examine only the large-scale constructs of the program. For example, you might first be interested in understanding the relationships between the modules that comprise the system.

To discover the modular structure of the FORTRAN system, you may direct the Packager component of the RET prototype to *cluster* subprograms into groups that will eventually become Ada packages. The Packager iteratively applies clustering techniques described by Hutchens (Ref. 9) and Muller (Ref. 10) to analyze the FORTRAN system and group subprograms based upon calling relationships and patterns of data usage, measured in terms of data bindings.

During the analysis, you may gain a better understanding of each subprogram's purpose and why subprograms are grouped as they are. The Packager invites you to explore the most recently clustered modules after each iteration. The RET prototype organizes the subprograms and helps you explore groups of related subprograms that are, in the sense of the clustering criteria, more closely related than others.

4.1.1 Definitions

The Packager uses data bindings to cluster some modules. A *data binding* is a tuple (p, x, q) where p and q are subprograms that reference data object x. The RET only counts data bindings in which the data object is written by one subprogram and read by the other.

The RET computes the Interconnection Strength (IS) and the Common Client and Supplier (CS) sets based upon the actual data bindings. The IS is the number of data bindings between two subprograms. Wherever the RET prototype displays the IS between two subprograms, it adds one to it if either of the two subprograms calls the other.

The common clients of a group of subprograms are those subprograms which read data that is written by every subprogram in the group. The common suppliers of a group of subprograms are those subprograms that provide data to every subprogram in the group.

Clustering produces a *tree of modules*. The root module represents the Ada library, intermediate modules represent packages, and the leaves represent subprograms. The root is defined to be at level zero and its children are at level one.

The Packager displays one graph for the Ada library, and one for each potential package. The Packager graph is an abstraction that presents relationships among program modules, such as data objects shared between them. *Nodes in the graph* represent nested modules, i.e., packages or subprograms, and the edges depict data binding relationships between them.

There is exactly one graph associated with any given nonleaf module, M, in the tree; we refer to it as *the* graph of M. The nodes in that graph correspond to the direct children of M in the tree. The edges between the nodes in the graph depict the data binding relationships between the corresponding packages or subprograms. We use the term *package structure* to refer to both the tree and its graphs. For simplicity, we refer to nodes as packages or subprograms or, when the distinction is unnecessary, as modules.

The graph in Figure 4 shows one subprogram *(FCR_OUTPUT)*, five packages *(fcr_df, fcr_dr, main, modes,* and *dead code)*, nine edges, and nine edge labels. An edge between two modules indicates that they share data bindings. An edge between a package and another module, M, indicates that at least one subprogram in the package shares data bindings with M. The edge labels list the data objects that comprise the bindings or show the subprograms in the package that are involved in the data bindings.

15
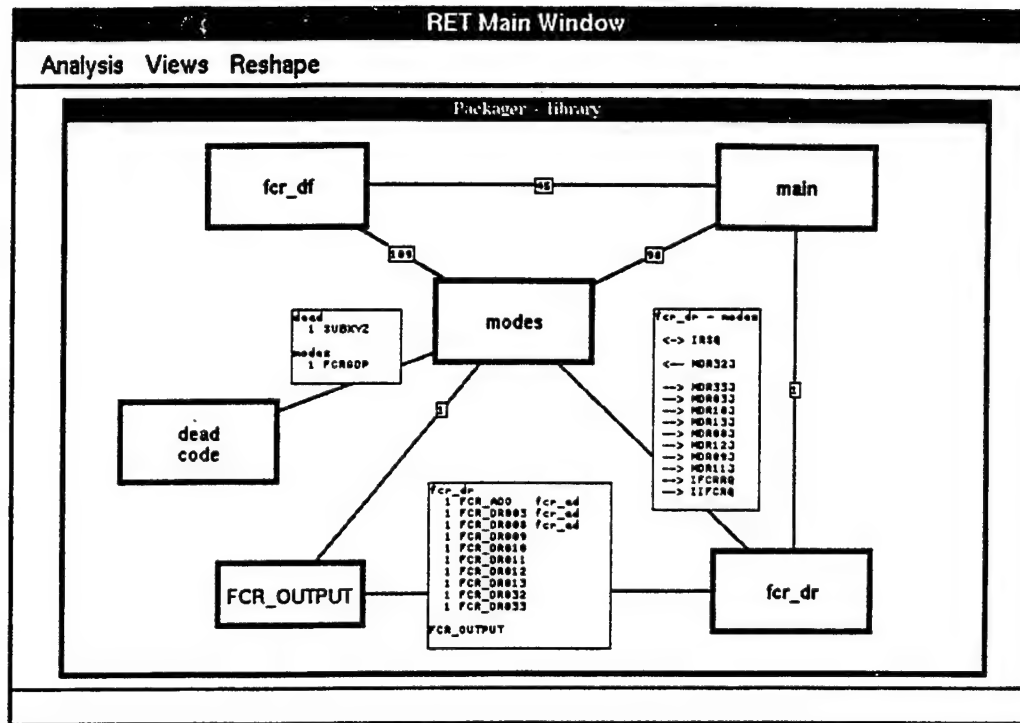
Figure 4   The Packager View

The edge labels in Figure 4 provide information about the variables shared between the modules. The edge label between packages *modes* and *fcr_dr* lists the variable names involved in the data bindings between them. This label lists one variable (*IRSQ*) that is read and written by both packages, one variable (*MDR32J*) that is read by *modes* and written by *fcr_dr*, and ten variables that are read by *fcr_dr* and written by *modes*.

The edge label between *FCR_OUTPUT* and *fcr_dr* shows that there is one data binding between *FCR_OUTPUT* and each of seven subprograms nested directly under *fcr_dr*, and one data binding between *FCR_OUTPUT* and each of three subprograms (*FCR_AD0*, *FCR_DR003*, and *FCR_DR008*) nested directly under *fcr_ad*, which is nested directly under *fcr_dr*.

The label between *dead code* and *modes* is similar, except that it is between two packages. The label on the edge between *FCR_OUTPUT* and *modes* shows the total number of variables (1) shared between them.

4.1.2   Creating a Package Structure

Initially, every subprogram is at level one in the package structure and appears in the level-zero graph. The edges in the graph are not shown by default because there are

16

generally too many of them, but you may view (or hide) the edges adjacent to any module by choosing the appropriate menu option. You may view the original source code associated with any module by selecting it with the mouse. You may select specific subprograms to be grouped together, or apply an automatic clustering algorithm.

The RET provides two clustering metrics and each can be defined as a function of two subprograms. The *Common Clients and Suppliers* (CS) metric counts the number of other subprograms that provide data to, or accept data from two subprograms. This metric is useful for locating and grouping utility or library routines, such as math or I/O routines. The *Interconnection Strength* (IS) metric counts the number of shared data items that two subprograms reference. This metric is useful in grouping subprograms that manipulate common global variables or exchange data by parameters.

The automatic clustering process is iterative. To begin clustering, we recommend that you direct the RET prototype to perform one clustering iteration using the CS metric. We call this strategy *CS-clustering*. It tends to group modules that receive data from, or pass data to the same modules. The Packager computes the common client and supplier sets for each pair of level-one modules and identifies the group of modules that share the greatest number of clients or suppliers. It then modifies the package structure to combine the modules in this group.

We recommend that you perform CS-clustering one iteration at a time for several reasons. CS-clustering only takes a few iterations to identify many of the utility subprograms, and the RET prototype relies on you to determine when to stop clustering. The RET also relies on you to manually add or remove subprograms because the heuristic strategy is imperfect.

Once you decide that CS-clustering is not uncovering any new utility subprograms, you may choose to initiate *IS-clustering*. With this strategy, the Packager computes the interconnection strength between each pair of level-one modules; determines the maximum IS, denoted IS_{max}, among all the modules; and groups those modules that are involved in an IS_{max} relation. You may perform IS-clustering one iteration at a time, but it is faster to direct the Packager to iterate until only one level-one module remains in the level-zero graph, i.e., all subprograms have been clustered into (possibly nested) packages.

We recommend that you employ CS-clustering before IS-clustering because the former identifies groups of utility subprograms that are not recognized by the latter. If IS-clustering combined a utility subprogram with other subprograms, the CS metrics for the

resulting package would be different from the utility subprogram's CS metrics and the utility would be less likely to combine with other utilities during CS-clustering.

With either clustering strategy, when the Packager groups a set of modules, it creates a new level-one package and moves the grouped modules to level two, i.e., under the new package. Edges appear in the level-zero graph between the new package and any level-one modules that share data bindings with it.

With either strategy, you may inspect and/or alter the package structure after each Packager iteration. Alternatively, you may direct the Packager to iterate until every module has been included in some package, automatically providing an approximation to a reasonable package structure. You should verify the resulting package structure because you may wish to modify the structure through the views to obtain an appropriate grouping. The information provided to you by the Packager facilitates this analysis, and editing operations allow you to easily change the structure.

The clustering strategies described above produce a hierarchical organization of packages; there are packages nested within other packages. Although the RET prototype can generate Ada code corresponding to a hierarchical nesting structure, it may be easier to maintain Ada code which consists of smaller library unit packages because such designs tend to discourage redundancy and strengthen encapsulation. You may wish to "flatten" the generated package structure, i.e., increase its width and decrease its depth. The RET prototype generates *with* context clauses for any package that references a library unit.

The Packager tries to prevent the package structure from becoming unnecessarily deep by maintaining a threshold on the package size. When package A is to be moved into package B such that A would be nested within B, the Packager checks the number of modules in package A. If it is below the threshold that you have specified, then the modules in A are moved to B and the package A is eliminated.

This somewhat arbitrary heuristic is only useful for preventing the formation of many tiny packages and, in practice, the threshold must be set quite low. The threshold is set to five in the RET prototype. You may wish to intervene during clustering and edit the package structure as it evolves in order to reduce nesting.

4.1.3 Editing the Package Structure

Packager graphs are interactive displays. You may open pop-up menus by positioning the mouse cursor over a module, an edge, the background, or the window title and

clicking the right mouse button. The resulting pop-up menu shows commands for the module, edge, background, or window. We refer to this below as issuing a module, edge, background, or window command. The RET provides commands for *navigating*, *browsing*, and *editing* the package structure.

Navigation commands allow you to display different graphs by clicking on a package or the background. The descend package command causes the RET to display a package's graph. The ascend background command causes the RET to display the parent package's graph.

Browsing commands alter the Packager display without changing the generated package structure. The RET prototype provides browsing commands on the module, edge, background, and window pop-up menus.

- Module commands allow you to select or deselect individual modules or modules in a region, show or hide individual or selected modules, drag and reshape modules, and show FORTRAN and/or Ada source code.

- Edge commands allow you to select or deselect individual edges, show or hide individual or selected edges, and show global or local bindings (or both) on edges.

- Background commands allow you to arrange modules in a circle or grid, and refresh, scroll, and zoom the display.

- Window commands allow you to move, refresh, hide, reshape, and close the Packager display.

Editing commands allow you to edit module names and alter the Ada package structure. You may move a module from the current graph to another package in that graph via the push command, or to the parent package's graph via the pop command. The pop-to-top command moves a package all the way up to the Ada library level. The disperse command eliminates a package from the current graph and moves all of the modules that were nested in it up one level in the graph. The RET maintains the edges between the packages and subprograms as you change the package structure.

You can assign navigation and source code display commands to the middle mouse button to reduce the number of mouse or keyboard events required to effect a command. We have found this to be very convenient when working with a large system. The left mouse button is always assigned to the select and deselect commands. The right mouse button is always assigned to the pop-up-menu command.

4.1.4 Distributing Data Items

The Packager automatically distributes global data items among the modules of the package structure. The algorithm reduces each data item scope while maintaining its visibility as needed. It is based upon the following criteria.

- If a data item is used only by subprograms in a single package, the data item declaration is placed in the package body.

- If a data item is used by subprograms in more than one package, but most often by subprograms in a particular package, the data item declaration is placed in that package specification. Other packages that use the data item specify a context clause for the package.

- A new package is created for each common block with remaining undistributed data items. These data items are used by subprograms in more than one package, with no package clearly using them more often. The data item declarations are placed in the new package specifications, and other packages that use the data items specify context clauses for the new packages.

The data object distribution algorithm is most effective when there are many data objects declared in one module, such as a common block, that are referenced by few other modules. Embedded systems may use common blocks to map variables to specific memory locations. **Warning: distributing these variables among the packages so as to reduce the scope of their declarations would disperse the specification of the mapping throughout the code and make it more difficult to change the mapping.**

The RET prototype analyzes FORTRAN *EQUIVALENCE* statements to check for memory-mapped common blocks. The RET only distributes the variables from common blocks that it determines not to be memory-mapped. The RET prompts you to accept or override it's choice before it distributes any variables.

4.2   TRANSLATING PROGRAM STATEMENTS

Once you have constructed and refined the package structure of the Ada system and placed the variable declarations where desired, the Transformer component of the RET prototype helps with micro restructuring by translating individual statements from the source to the target programming language. You may inspect the FORTRAN Source Code Listing view for any module and select statements with the mouse. The RET translates them to Ada and inserts them into the Ada ASG. If you have renamed variable declarations, then the RET prototype generates references to those variables.

The RET prototype generates Ada code for the package structure that the Packager produces. First the RET creates a skeleton of the Ada system, and then it transforms individual statements. The skeleton Ada code comprises package and subprogram specifications and bodies that may include variable and constant declarations. The subprogram bodies include a single *null* statement. The RET generates subunits at your option. The RET also generates a type package that defines all of the types and subtypes referenced in the variable declarations. The type package declares Ada types that correspond closely with FORTRAN types, although an exact mapping is not generally available as explained below.

Translating FORTRAN statements that map readily onto Ada language features is straightforward. For example, the RET prototype can easily translate *Block IF* and *DO/END DO* statements into Ada IF and LOOP statements, respectively, because the semantics are consistent between the languages. The fact that the control variable of a FORTRAN *DO* statement remains defined after the loop is a nuisance.

There are FORTRAN constructs for which the mapping to Ada is not obvious or for which there is a choice of translations. We have found several sources of difficulty in transforming individual statements in such a way as to avoid propagating undesirable FORTRAN constructs while taking advantage of Ada language features not present in FORTRAN. They include the use of *unstructured control constructs*, the general lack of correspondence between *language features* and, in particular, differences in the *data type systems*.

4.2.1   Unstructured Control Constructs

Some FORTRAN code contains unstructured control forms, defined simply as branches into or out of loops or decisions. While such forms do not *always* impede maintenance, they usually make the code harder to understand and modify. Unstructured control forms exist in code that was written before the benefits of structured programming were widely acknowledged.

Some FORTRAN language features encourage unstructured designs. *Arithmetic IF* statements cause control to be transferred to any one of three locations based on a test. *Logical IF* statements are only problematic when they are used with *GOTO* statements. VAX FORTRAN extended ranges (in *DO* loops) are egregious examples of unstructured constructs that might effectively confound maintenance programmers. The RET prototype assumes that you have processed the FORTRAN code with SPAG (Ref. 3), a commercial control flow restructuring tool, to eliminate control structures that are difficult to translate. The tool removes most of the objectionable constructs.

4.2.2   Language Features

The RET prototype generates code for Ada language features that have no counter-part in FORTRAN, but which produce programs that are substantially easier to maintain. For example, the RET does take advantage of Ada packages because we feel that they are useful for encapsulating code and help to reduce the ripple effect of modifications. On the other hand, the RET prototype does not generate Ada code which uses exceptions because we believe that they make the code more difficult to understand and, except in select situations, are of limited value.

4.2.3   Data Type Systems

FORTRAN has fewer types than Ada and it allows implicit conversions which must be explicit in Ada. Data types that seem to serve the same purpose may have different implementations across languages. The application may even rely upon compiler implementation details or undocumented language features. This fact is often an important consideration when translating embedded systems. The ASRET Final Report, Vol. I (Ref. 1) provides details on how the RET prototype converts data types.

5.   RET REFERENCE

This section provides reference information for the RET prototype user interface. It assumes that you know how to start the prototype. If you don't, ask the person who installed it because several options may be available. If you are responsible for installing the RET prototype, see The Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume I, Project Summary, Account, and Results (Ref. 1). Section 8 of Vol. I, RET PROTOTYPE PLATFORM, provides information on running the RET prototype under GNU Emacs (Ref. 11). You may alternatively wish to create an executable to run stand-alone, i.e., without Emacs.

The RET prototype assumes that the FORTRAN source code was processed by SPAG (Ref. 3) and REFINE/FORTRAN (Ref. 12) as described in the ASRET Final Report (Ref. 1). The input to the RET prototype (Section 5.1) is the analysis file created by REFINE/FORTRAN.

The reference information in this section is organized around the RET prototype views. Each section explains how to operate a single view (Table 2). The reference assumes that you are familiar with common Graphical User Interface (GUI) concepts such as windows, the mouse cursor and buttons, and pull-down and pop-up menus. The discussions on the menu options listed in Table 3 through Table 25 explain only those features of the RET prototype that are not ubiquitous in GUIs.

Table 2   RET Reference Directory

SECTION	DESCRIBES
5.1	RET main window
5.2	Packager view
5.3	Dataflow Diagram
5.4	Call Diagram
5.5	Declaration Diagram
5.6	FORTRAN Source Code Listing
5.7	Ada Source Code Listing

5.1 RET MAIN WINDOW

The Main Window in Figure 5 shows the title area at the top, the command area beneath it, and the workspace with three views and one pop-up window. The Packager view is partially occluded by the PACKAGE VARIABLES pop-up window and the Declaration Diagram (DED) and Call Diagram (CD) views. The command area lists the **Analysis**, **Views**, and **Reshape** pull-down menus. You may activate the pull-down menus by clicking on them with, and holding down the left mouse button, an operation referred to as "left-clicking."

To begin an analysis, pull down the **Analysis** menu shown in Figure 6 by left-clicking on **Analysis** in the command area. While holding down the left mouse button, drag the cursor over the **Perform** menu option, and then release the left mouse button. The RET prototype pops up a window that prompts for the REFINE/FORTRAN analysis file. You may edit the file name in this window as explained in the INTERVISTA User's Manual (Ref. 13).

After you enter the file name, the RET prototype reads the analysis file and performs initialization processing. This may take up to an hour for a system with about twenty thousand lines of code. While the RET is processing the analysis file, it will print messages to the Emacs *REFINE* buffer. When initialization is complete, the RET will print the message "Analysis Complete."
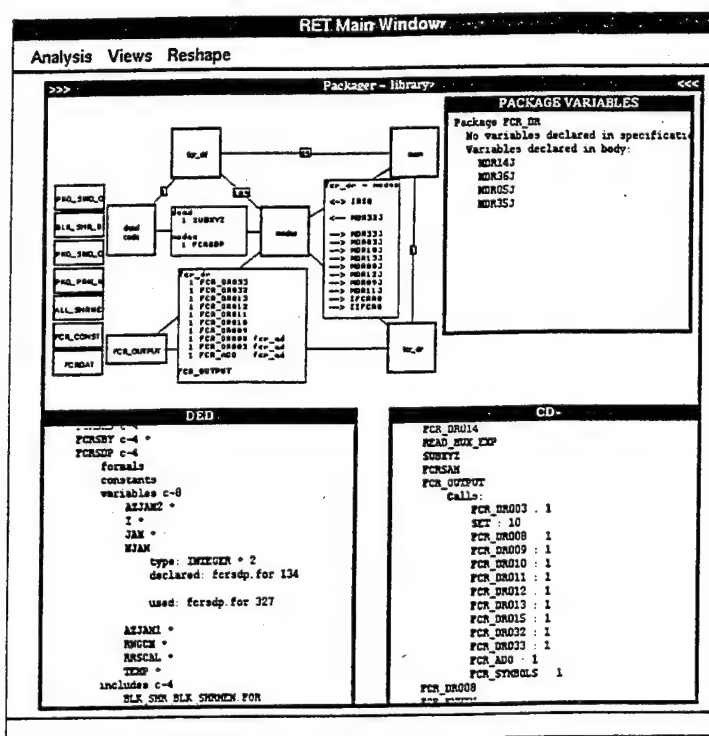
Figure 5   RET Main Window
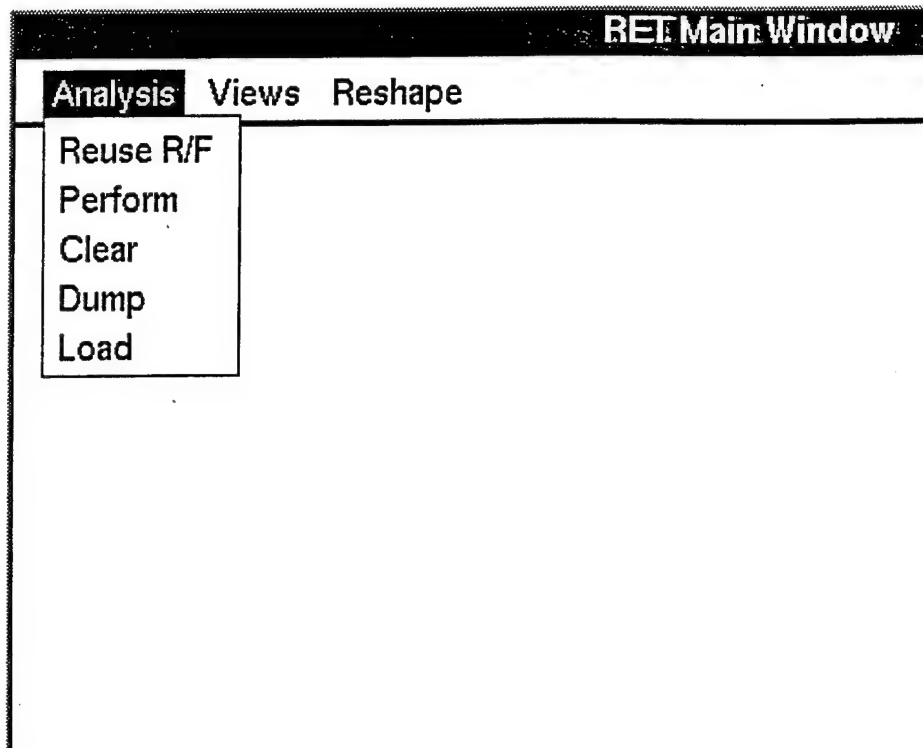
Figure 6   Analysis Pull-Down Menu

The **Reuse R/F, Clear, Dump**, and **Load** menu options shown in Figure 6 are no longer in use, but remain in the pull-down menu because future versions of the RET prototype may use them.

You may now show the DED, CD, or Packager views by selecting the **Show DED**, **Show CD**, or **Show Packager** menu options, respectively, shown in Figure 7. The RET prototype can not produce a DFD until it generates Ada source code, so don't select the **Show DFD** or **Show Ada Source** options at this time. You may select the **Regenerate** options at any time to cause the RET prototype to recreate one of the views. You may load a DFD or Packager view if you have previously saved one by selecting the **Load DFD** or **Load Packager** option.

The **Reshape** pull-down menu in Figure 8 provides options to change the size and position of the "current" view. Only one view at a time is designated as the current view. The title of the current view is enclosed by ">>>" and "<<<" as shown in Figure 5, wherein the Packager view (the window entitled "Packager - library") is current. The DED and CD views in Figure 5 were reshaped via the **Lower Left** and **Lower Right** menu options, respectively.

```
┌─────────────────────────────────────────────────┐
│                               RET Main Window    │
├─────────────────────────────────────────────────┤
│ Analysis │ Views │ Reshape                        │
│          ├───────────────────────┐               │
│          │ Show DED              │               │
│          │ Regenerate DED        │               │
│          │ Show CD               │               │
│          │ Regenerate CD         │               │
│          │ Show FORTRAN Source   │               │
│          │ Show Ada Source       │               │
│          │ Load DFD              │               │
│          │ Show DFD              │               │
│          │ Regenerate DFD        │               │
│          │ Load Packager         │               │
│          │ Show Packager         │               │
│          │ Regenerate Packager   │               │
│          └───────────────────────┘               │
│                                                   │
```
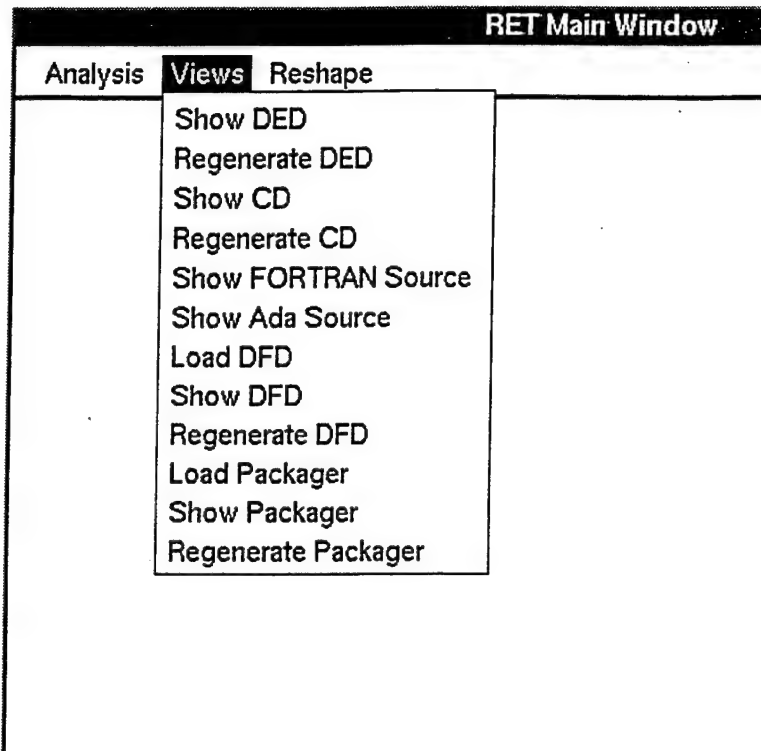
Figure 7   Views Pull-Down Menu

```
┌─────────────────────────────────────────────────┐
│                           RET Main Window        │
├─────────────────────────────────────────────────┤
│ Analysis │ Views │ Reshape                        │
│                  ├──────────────┐                │
│                  │ Full Screen  │                │
│                  │ Upper Half   │                │
│                  │ Lower Half   │                │
│                  │ Left Half    │                │
│                  │ Right Half   │                │
│                  │ Upper Left   │                │
│                  │ Upper Right  │                │
│                  │ Lower Left   │                │
│                  │ Lower Right  │                │
│                  └──────────────┘                │
│                                                   │
```
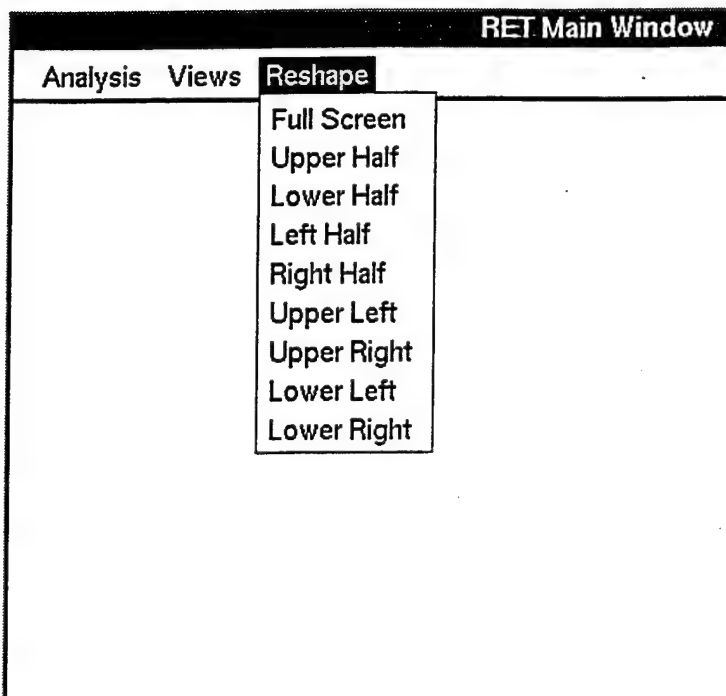
Figure 8   Reshape Pull-Down Menu

26

5.2 PACKAGER VIEW

Figure 9 is a duplicate of Figure 4 and is explained in Section 4.1.1. Figure 10 shows a Packager view with Call edges that depict subprogram calls. The edge on the left indicates that subprogram RLT_INPUT calls RLT_F066_IN and RLT_REF_IN, both of which are declared in package rlt_in. The edge on the right shows, in a different format, that RLT_OUTPUT calls RLT_RLTADO and RLT_RL001_OUT. The RET prototype provides the second format because the first does not distinguish individual subprogram calls when the edge appears between two packages. You may alternate between the two formats by middle-clicking on the edge label.

Figure 11 shows a Packager view containing five intrinsic functions generated by the RET prototype and the corresponding generated Ada Source Code Listing for the INTRINSICS package. The formal parameter types and the return types are given in the nodes.

Figure 12 shows a Packager view containing four external subprograms generated by the RET prototype and the corresponding generated Ada Source Code Listing for the EX-TERNALS package. The formal parameter types and the return types (for the function) are given in the nodes. Note that external subprograms may be functions or procedures.
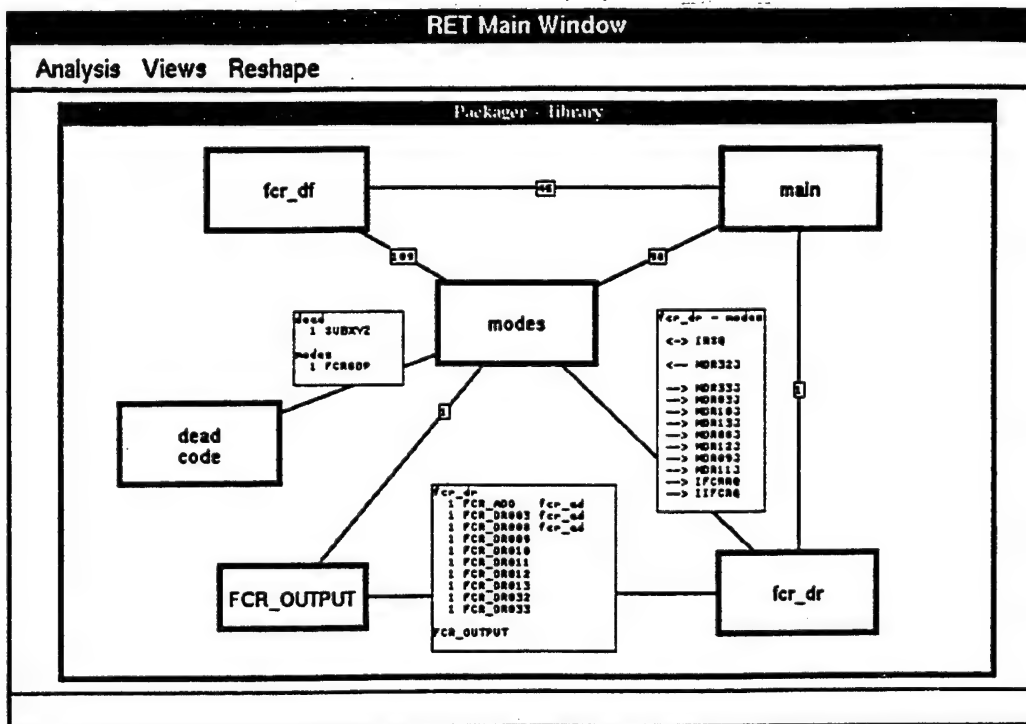
Figure 9   Packager View With Data Binding Edges

27
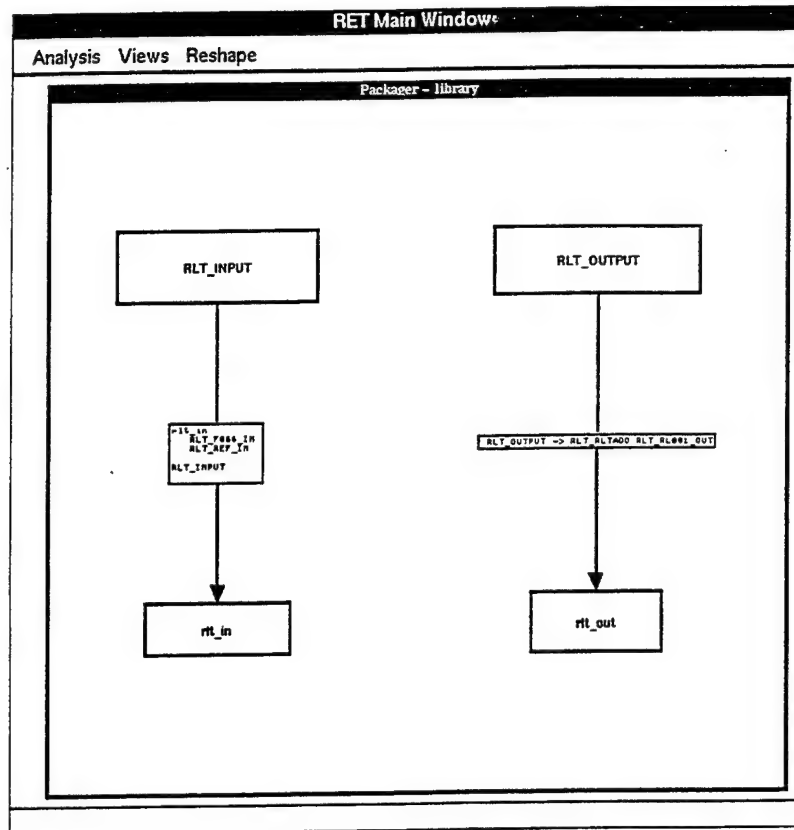
Figure 10   Packager View With Call Edges
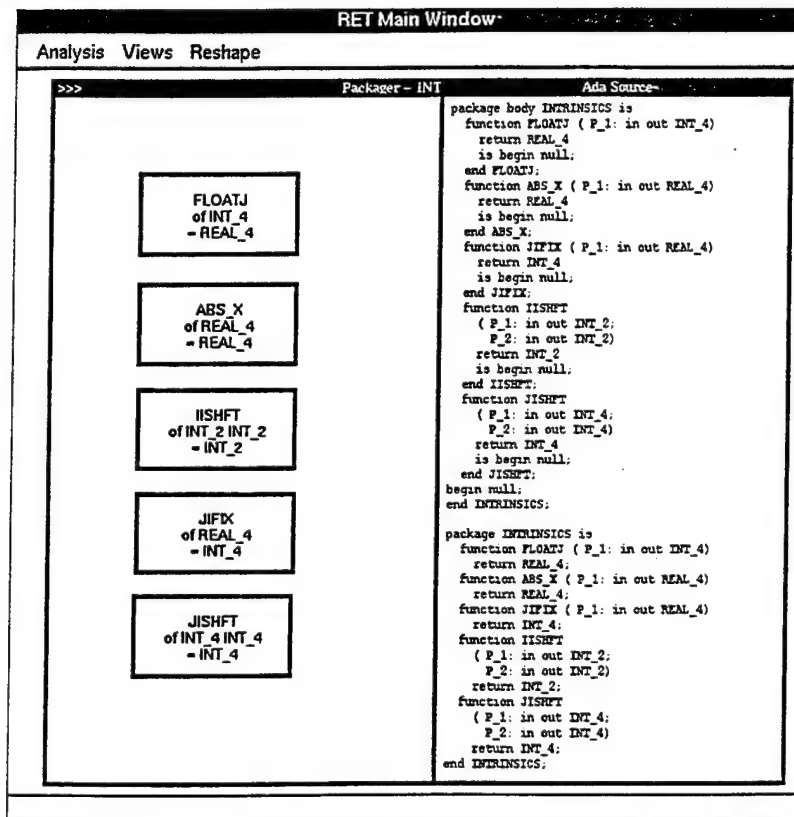
Figure 11   Packager and Source Code Views of Intrinsic Functions
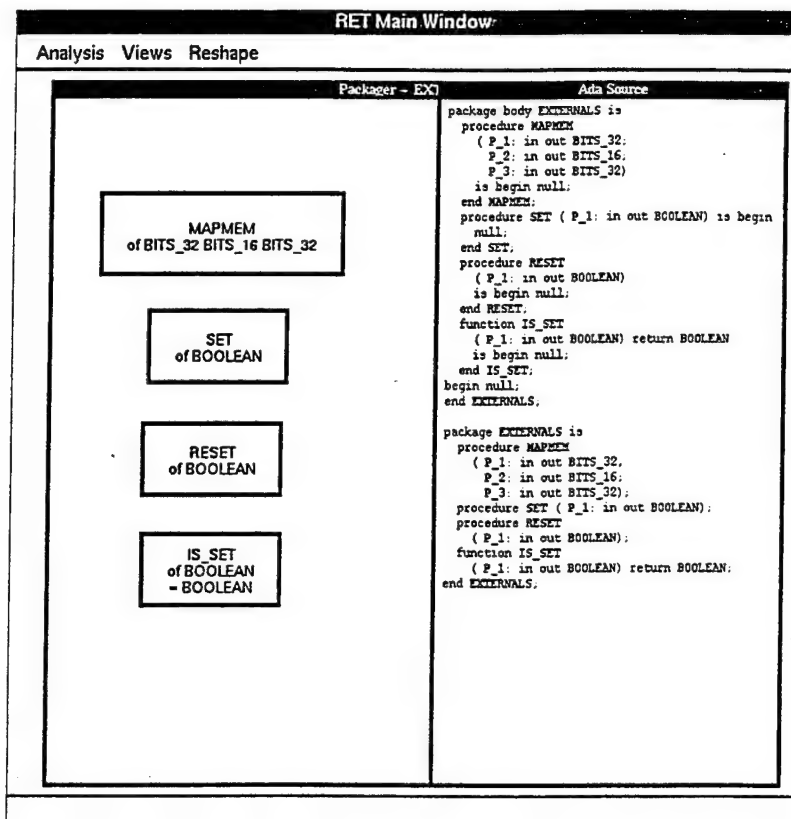
Figure 12 Packager and Source Code Views of External Subprograms

Table 3 lists the options in the menu that pops up when you right-click on a package node in the Packager view. The **Drag** options allow you to reposition a single node (also called an icon), all selected nodes, or all nodes that have an edge to the current node by left-clicking. **Reshape** lets you change the size of a node. **Generate Statements** causes the RET to transform the FORTRAN code for the current node to Ada. **Write Files** creates ASCII files for the current node, and **Parse Files** parses them without analyzing them. **Edit Name** lets you rename a node. **Edit Spec** and **Body** open Emacs buffers for the specification and body files, respectively. **(De)Select** Icon changes the reverse-video status of a node, making it (in)sensitive to other menu options that operate on selected nodes.

Table 3 Packager Package Pop-Up Menu

PAK: PACKAGE
Drag
Drag Selected Icons
Drag Connected Icons
Reshape
Generate Statements
Write Files
Parse Files
Edit Name
Edit Spec
Edit Body
Select Icon
Deselect Icon
Mouse Left = Select
Mouse Left = Drag
Mouse Middle = View FORTRAN Code
Mouse Middle = View FORTRAN/Ada Code
Mouse Middle = View Ada Code
Mouse Middle = Navigate
Mouse Middle = Reshape
Hide Icon
Hide Edges
Show Edges
Draw Edges Manually
Draw Edges Automatically
Draw Edges as Straight Lines
Show Data Bindings
Show Call Relations
Show Ada Source Code
Show Package Variables
Set Subprograms Separate False
Push Selected Icons Into Package
Push Into Package
Pop Out of Package
Pop to Library
Disperse
Delete
Descend
PUP
Inspect
Abort

The **Mouse Left (Middle)** options assign the specified functions to the left (middle) mouse buttons. The **Navigate** function implies **Ascend** (Table 6) and **Descend**. The other functions are explicitly listed as menu options. **Hide Icon (Edges)** causes the objects to disappear. **Show Edges** causes all hidden edges to reappear. The **Draw Edges** options provide a means to respecify the layout of the edges. **Show Data Bindings** and **Show Call Relations** alternate the types of edges that are shown. **Show Package Variables** pops up a window that lists the variables in the package. **Set Subprograms Separate True (False)** (re)sets flags that cause the Ada code for the subprograms to be generated as subunits. The **Push** and **Pop** options allow you to move nodes down and up, respectively, in the package hierarchy. **Disperse** dissolves the package and promotes its nested nodes to the current graph. **Descend** causes the package's graph to replace the current graph. **PUP** and **Inspect** are debugging options. **Abort** closes the menu without taking any action.

Table 4 lists the options in the menu that pops up when you right-click on a subprogram node in the Packager view. The **Drag** options allow you to reposition a single node, all selected nodes, or all nodes that have an edge to the current node. After choosing the Drag option for a node, move the mouse to a clear area on the background while holding down the left mouse button and release it at the desired position. **Reshape** lets you change the size and shape of a node. **Generate Statements** causes the RET to transform the FORTRAN code for the current node to Ada. **Write Files** creates ASCII files for the current node, and **Parse Files** parses them without analyzing them. **Edit Name** lets you rename a node. **Edit Spec** and **Body** open Emacs buffers for the specification and body files, respectively. **(De)Select** Icon changes the reverse-video status of a node, making it (in)sensitive to other menu options that operate on selected nodes.

Table 4 Packager Subprogram Pop-Up Menu

PAK: SUBPROGRAM
Drag
Drag Selected Icons
Drag Connected Icons
Reshape
Generate Statements
Write Files
Parse Files
Edit Name
Edit Spec
Edit Body
Select Icon
Deselect Icon
Mouse Left = Select
Mouse Left = Drag
Mouse Middle = View FORTRAN Code
Mouse Middle = View FORTRAN/Ada Code
Mouse Middle = View Ada Code
Mouse Middle = Navigate
Mouse Middle = Reshape
Hide Icon
Hide Edges
Show Edges
Draw Edges Manually
Draw Edges Automatically
Draw Edges as Straight Lines
Show Data Bindings
Show Call Relations
Show FORTRAN Source Code
Show FORTRAN/Ada Source Code
Show Ada Source Code
Push Into Package
Pop Out of Package
Pop to Library
Disperse
Delete
Descend
PUP
Inspect
Abort

32

Table 5 lists the options in the menu that pops up when you right-click on a data bindings edge label in the Packager view. The **Show Call** and **Show Binding** options change the type of edge label that the Packager displays on call and data binding edges, respectively. The **Show Global** and **Local** options affect whether only global variables or just local variables and parameters, respectively, are included in the edge labels. **Hide Edge** causes the edge to disappear. The **Draw Edge** options are for respecifying the edge layout. The **Mouse Left (Middle)** options assign the specified functions to the left (middle) mouse buttons. The **Draw** functions correspond to the Draw menu options. The **Toggle Display** function changes the type of edge label that is shown on the edge. **PUP** and **Inspect** are debugging options. **Abort** closes the menu without taking any action.

Table 5 Packager Data Binding Edge Pop-Up Menu

PAK: BINDINGS
Show Call Total (CS)
Show Call Summary
Show Call Details
Show Binding Total (IS)
Show Binding Summary
Show Binding Details
Show Global Bindings Only
Show Local Bindings Only
Show Source Code
Hide Edge
Draw Edge Manually
Draw Edge Automatically
Mouse Left = Select
Mouse Left = Drag
Mouse Middle = Draw Auto
Mouse Middle = Draw Manual
Mouse Middle = View Source
Mouse Middle = Toggle Display
PUP
Inspect
Abort

Table 6 lists the options in the menu that pops up when you right-click on the background in the Packager view. The **Cluster** options invoke the algorithms that combine nodes into packages. **Create Package** creates a new package node with no declarations. The **(De)Select** options change the reverse-video status of nodes, making them (in)sensitive to other menu options that operate on them. The **Show** options for **Icons** and **Edges** cause hidden nodes and edges, respectively, to reappear. The **Show** options for **Data Bindings** and **Call Relations** cause the specified kinds of edges to appear for **All** nodes, or just for **Selected** nodes. The **Hide** options cause nodes and edges to disappear. The **Push** and **Pop** options let you move nodes down or up, respectively, in the package hierarchy.

Table 6 Packager Background Pop-Up Menu

PAK: BACKGROUND
Cluster by IS Once
Cluster by IS Until Done
Cluster by CS Once
Cluster by CS Until Done
Create Package
Select All Packages
Select All Subprograms
Select Icons in Region
Deselect Icons in Region
Deselect All
Show Hidden Icons
Show Hidden Edges
Show Hidden Icons/Edges
Show Edges of Selected Icons
Show All Data Bindings
Show All Call Relations
Show Selected Data Bindings
Show Selected Call Relations
Hide All Edges
Hide Edges of Selected Icons
Hide Selected Icons/Edges
Push Selected Icons
Pop Selected Icons
Pop Selected Icons to Top

Table 6 Packager Background Pop-Up Menu (Continued)

PAK: BACKGROUND
Drag
Ascend
Refresh View
Draw All Edges Automatically
Draw All Edges as Straight Lines
Neutral Scroll
Normal Scale
Zoom In
Zoom Out
Arrange Icons in Circle
Arrange Selected Icons in Circle
Arrange Icons in Grid
Arrange Selected Icons in Grid
Mouse Left = Select
Mouse Left = Select
Mouse Left = Drag
Mouse Middle = View Source
Mouse Middle = Navigate
PUP
Inspect
Inspect

Drag moves a node. After choosing the Drag option for a node, move the mouse to a clear area on the background while holding down the left mouse button and release it at the desired position. Ascend replaces the current graph with the parent package graph. The Draw All Edges options let you respecify the layout for all edges in the graph. Normal Scroll changes the magnification of the current view to a preset level. Neutral Scale changes the magnification of the current view so that all nodes are shown. The Arrange Icons options let you move all or selected nodes so that they form a circle or grid. The Mouse Left (Middle) options assign the specified functions to the left (middle) mouse buttons. The Navigate function implies Ascend and Descend (Table 3). The other functions are explicitly listed as menu options. PUP and Inspect are debugging options. Abort closes the menu without taking any action.

Table 7 lists the options in the menu that pops up when you right-click on the window title bar of the Packager view. The **Window** options let you control the Packager window. **Save Package Structure** pops up a window to prompt you for a file name in which to save the internal Packager representation. You can subsequently reload the file by selecting the **Load Packager** option in Figure 7. The **Delete Package Structure** clears the current Packager view. You select the following options, in the specified order, to generate Ada code after you are satisfied with the package structure. (You can change the structure and regenerate code as often as you like.)

1. **Distribute Global Data Items**
2. **Initialize Ada Library**
3. **Generate Ada Statements**
4. **Generate Ada Code For Implicits**
5. **Write Ada Files**
6. **Analyze Ada Files**

PUP and Inspect are debugging options. Abort closes the menu without taking any action.

Table 7 Packager Window Pop-Up Menu

PAK: WINDOW
Cluster by IS Once
Lower Window
Hide Window
Move Window
Reshape Window
Reshape Window
Refresh Window
Print Window
Save Package Structure
Delete Package Structure
Distribute Global Data Items
Initialize Ada Library
Generate Ada Statements
Generate Ada Code For Implicits
Write Ada Files
Analyze Ada Files
PUP
Inspect
Abort

36

5.3   DATAFLOW DIAGRAM

Figure 13 shows a sample top-level Dataflow Diagram (DFD). It contains six transform nodes: RLT_INIT, RLT_INPUT, RLT_OUTPUT, and RLT_COMPUTE, RLT_TERM, and RLT_SUSPEND. The rectangular nodes are buffer repositories. The arrows indicate the direction of data flow between transform and buffer nodes.

Figure 14 shows the same top-level graph with most of the nodes selected (shown in reverse-video). Note that we have moved the CURRENT_POWER buffer and directed the DFD to display its edges, which were hidden in Figure 13. We captured the screen output which is Figure 14 when the mouse cursor was over the RLT_OUTPUT transform. The DFD outlines the node that the cursor is currently on and all adjacent nodes, i.e., nodes that share an edge with the current node. In this case, all nonselected nodes except for RLT_RL003_OUT and RLT_RL004_OUT are outlined because they are adjacent to RLT_OUTPUT. This feature comes in handy with more complicated graphs or when some edges are hidden.

We selected most of the nonadjacent (to RLT_OUTPUT) nodes because we want to focus on the RLT_OUTPUT transform. Figure 15 shows the result of hiding the selected nodes, zooming in, and reshaping the PREVIOUS_POWER node to reveal its entire label.
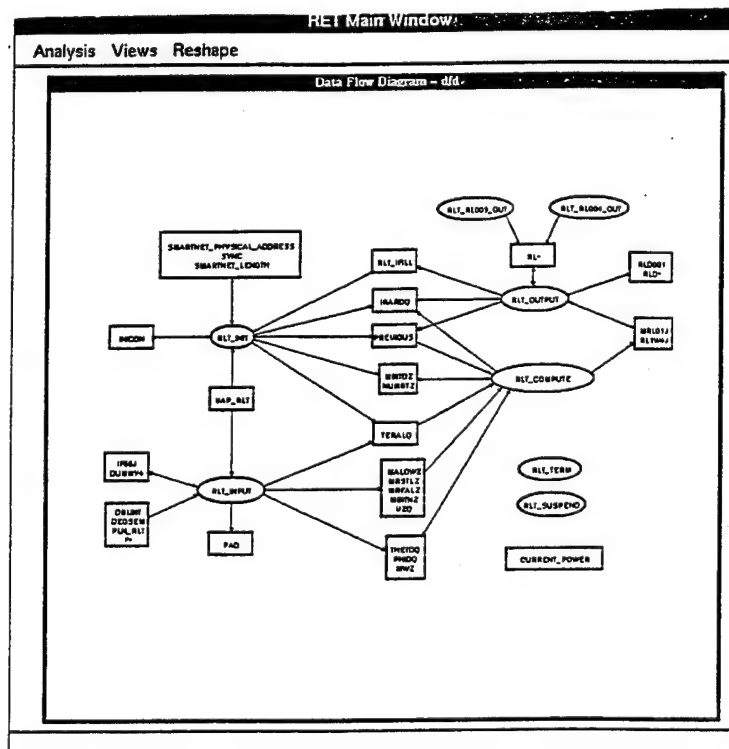
Figure 13   Dataflow Diagram Top-Level View

37
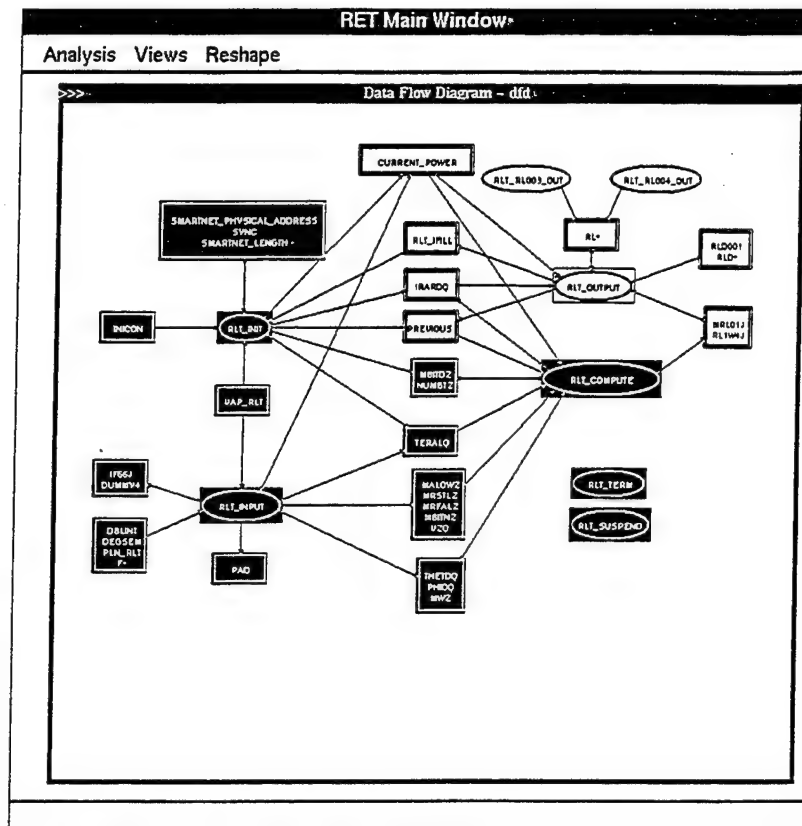
Figure 14   Dataflow Diagram With Selected Nodes
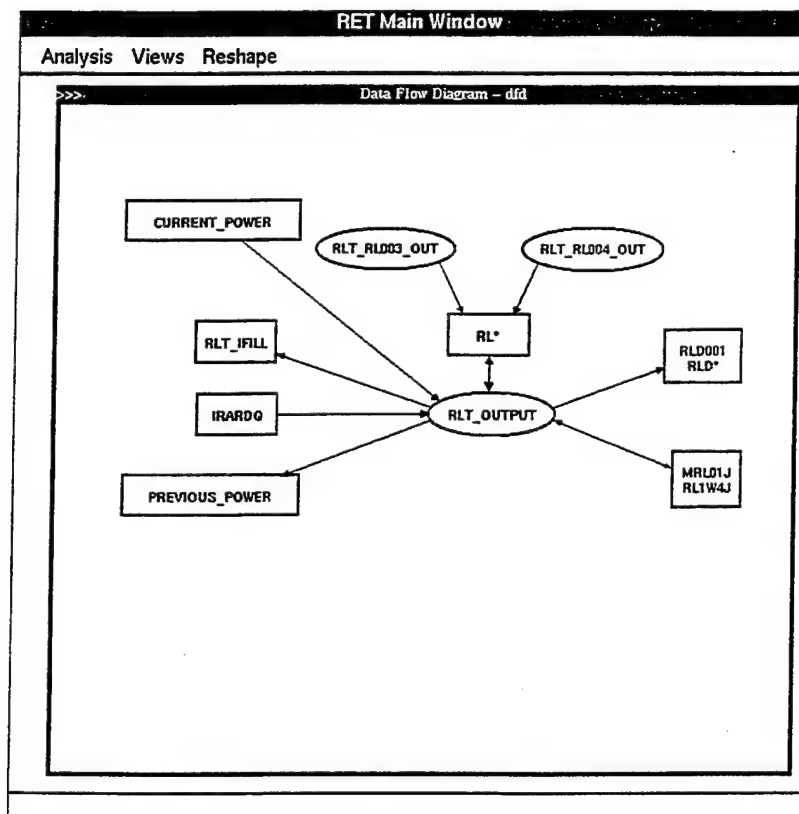
Figure 15   Dataflow Diagram With Hidden Nodes and Edges

38

The RLT_RL00n_OUT transforms are not directly relevant to RLT_OUTPUT and we could have hidden them too, but we did not.

Figure 16 shows the effect of descending into the RLT_OUTPUT transform. The RET has replaced the top-level graph with that of the RLT_OUTPUT transform node. This is accomplished by middle-clicking on the RLT_OUTPUT node in Figure 15, or right-clicking on it and choosing the **Descend** option from the resulting pop-up menu (Table 8).

Figure 16 depicts the same transformation as Figure 15, but in more detail. For example, Figure 15 shows that RLT_OUTPUT transforms IRARDQ into RLD001 and RLD*, but Figure 16 shows that it does this by calling RLT_RLTADO. On the other hand, statements in the body of subprogram RLT_OUTPUT transform CURRENT_POWER into PREVIOUS_POWER.

RLD001 represents a single variable, but RLD* represents a collection (or record) repository. A collection is a group of repositories that the DFD may form to reduce the number of repository nodes in a graph. Another way that the DFD reduces the number of repositories is to list more than one variable or collection in a single node, such as the one containing RLD001 and RLD* in Figure 16.
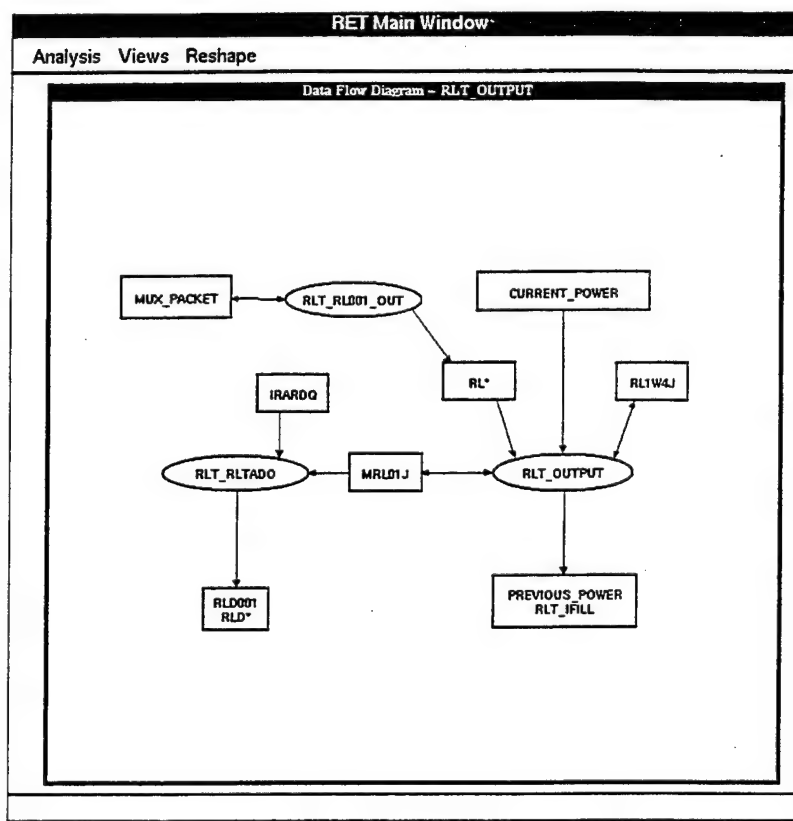
Figure 16 Dataflow Diagram for RLT_OUTPUT

Table 8 lists the options for menus that pop up when you right-click on call (non-terminal), leaf (terminal), or body transform nodes in the DFD view. **Descend** causes the package's graph to replace the current graph. **Reshape** lets you change the size and shape of a node. The **Drag** options allow you to reposition a single node, all selected nodes, or all nodes that have an edge to the current node. After choosing the Drag option for a node, move the mouse to a clear area on the background while holding down the left mouse button and release it at the desired position. **Edit Spec** and **Body** open Emacs buffers for the specification and body files, respectively. **(De)Select** Icon changes the reverse-video status of a node, making it (in)sensitive to other menu options that operate on selected nodes.

Table 8 Dataflow Diagram Pop-Up Menu for Transform Nodes

DFD: CALL NODE DFD: LEAF NODE DFD: BODY NODE
Descend
Reshape
Drag
Drag Selected Icons
Drag Adjacent Icons
Drag Connected Icons
Edit Spec
Edit Body
Select Icon
Select Adjacent Icons
Select Connected Icons
Deselect Icon
Deselect Adjacent Icons
Deselect Connected Icons
Mouse Left = Select
Mouse Left = Drag
Mouse Middle = View FORTRAN Code
Mouse Middle = View FORTRAN/Ada Code
Mouse Middle = View Ada Code
Mouse Middle = Navigate
Mouse Middle = Reshape
Hide Icon

**Table 8 Dataflow Diagram Pop-Up
Menu for Transform Nodes (Continued)**

DFD: CALL NODE DFD: LEAF NODE DFD: BODY NODE
Hide Edges
Show Edges
Show Local Variables
Show FORTRAN Source Code
Show FORTRAN/Ada Source Code
Show Ada Source Code
PUP
Inspect
Abort

The **Mouse Left (Middle)** options assign the specified functions to the left (middle) mouse buttons. The **Navigate** function implies **Ascend** (Table 10) and **Descend**. The other functions are explicitly listed as menu options. **Hide Icon (Edges)** causes the node (the node edges) to disappear. **Show Edges** causes all hidden edges to reappear. **Show Local Variables** pops up a window that lists the variables in the corresponding subprogram. **PUP** and **Inspect** are debugging options. **Abort** closes the menu without taking any action.

Table 9 lists the options for menus that pop up when you right-click on variable or record (collection) repository nodes in the DFD view. The description of the options is identical to that of Table 8.

**Table 9 Dataflow Diagram Pop-Up
Menu for Repository Nodes**

DFD: VARIABLE NODE DFD: RECORD NODE
Descend
Reshape
Drag
Drag Selected Icons
Drag Adjacent Icons
Drag Connected Icons
Edit Spec

**Table 9 Dataflow Diagram Pop-Up
Menu for Repository Nodes (Continued)**

DFD: VARIABLE NODE DFD: RECORD NODE
Edit Body
Select Icon
Select Adjacent Icons
Select Connected Icons
Deselect Icon
Deselect Adjacent Icons
Deselect Connected Icons
Mouse Left = Select
Mouse Left = Drag
Mouse Middle = View FORTRAN Code
Mouse Middle = View FORTRAN/Ada Code
Mouse Middle = View Ada Code
Mouse Middle = Navigate
Mouse Middle = Reshape
Hide Icon
Hide Edges
Show Edges
PUP
Inspect
Abort

Table 10 lists the options in the menu that pops up when you right-click on the background in the DFD view. **Ascend** replaces the current graph with the parent package graph. The **(De)Select** options change the reverse-video status of nodes, making them (in)sensitive to other menu options that operate on them. The **Show** options for **Icons** and **Edges** cause hidden nodes and edges, respectively, to reappear. The **Hide** options cause nodes and edges to disappear.

42

Table 10 Dataflow Diagram Pop-Up Menu for Background

DFD: BACKGROUND
Ascend
Select All Transform Nodes
Select All Call Nodes
Select All Leaf Nodes
Select All Data Nodes
Select Icons in Region
Deselect Icons in Region
Deselect All
Show Hidden Icons
Show Hidden Edges
Show Hidden Icons/Edges
Show Edges of Selected Icons
Hide All Edges
Hide Edges of Selected Icons
Hide Selected Icons/Edges
Drag
Refresh View
Neutral Scroll
Normal Scale
Zoom In
Zoom Out
Arrange Icons
Arrange Icons in Circle
Arrange Selected Icons in Circle
Arrange Icons in Grid
Arrange Selected Icons in Grid
Mouse Left = Select
Mouse Left = Drag
Mouse Middle = View Source
Mouse Middle = Navigate
PUP
Inspect
Abort

43

Drag moves a node. After choosing the Drag option for a node, move the mouse to a clear area on the background while holding down the left mouse button and release it at the desired position. Normal Scroll changes the magnification of the current view to a preset level. Neutral Scale changes the magnification of the current view so that all nodes are shown. The Arrange Icons options let you move all or selected nodes so that they form a circle or grid. The Mouse Left (Middle) options assign the specified functions to the left (middle) mouse buttons. The Navigate function implies Ascend and Descend (Table 8 and Table 9). The other functions are explicitly listed as menu options. PUP and Inspect are debugging options. Abort closes the menu without taking any action.

Table 11 lists the options in the menu that pops up when you right-click on the window title bar in the DFD view. The **Window** options let you control the DFD window. **Save DFD Structure** pops up a window to prompt you for a file name in which to save the internal DFD representation. You can subsequently reload the file by selecting the **Load DFD** option in Figure 7. The **Edit DFD Name** option lets you change the window title for the top-level DFD graph. The name is also the default file name for the **Save DFD Structure** option. **PUP** and **Inspect** are debugging options. **Abort** closes the menu without taking any action.

Table 11 Dataflow Diagram Pop-Up Menu for Window

DFD: WINDOW
Lower Window
Hide Window
Move Window
Reshape Window
Refresh Window
Print Window
Save DFD Structure
Edit DFD Name
Distribute Global Data Items
Initialize Ada Library
Generate Ada Statements
Generate Ada Code For Implicits
Write Ada Files
Analyze Ada Files
PUP
Inspect
Abort

5.4   CALL DIAGRAM

Figure 17 shows a sample Call Diagram (CD) as it appears immediately after generation. The "c-88" at the top indicates that the names of 88 subprograms follow. You may expand a subprogram by right-clicking on it and choosing the **Show Calls** or **Show Called By** pop-up menu options (see Table 12).

These and other options in the menus operate on the subprogram that you right-clicked on to pop-up the menu, i.e., the selected subprogram. You may also designate one or more selected subprograms by left-clicking on them. Left-clicking toggles the selected status of a subprogram. The CD distinguishes selected subprograms by displaying them in reverse video.

Figure 18 shows the CD after expanding FCR_DR013 to show the subprograms that call it (only FCR_OUTPUT calls FCR_DR013) and FCRS16 to show the subprograms that it calls (FCRIC, FCRSBY, etc.). The information that the CD displays for calls includes the subprogram names and the number of calls that the selected subprogram makes to each one. Figure 18 shows that FCRS16 calls FCRIC once and FCRMOD twice.

Figure 19 shows the same CD as Figure 18, except that it is positioned to start with FCRS16. You may expand any subprogram in the CD to show calling information.
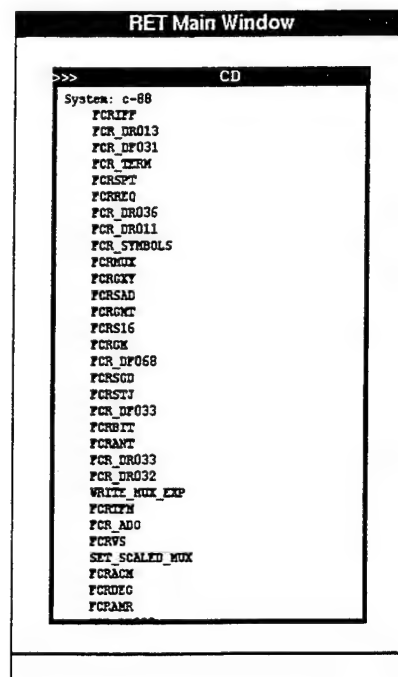
Figure 17   Initial Call Diagram
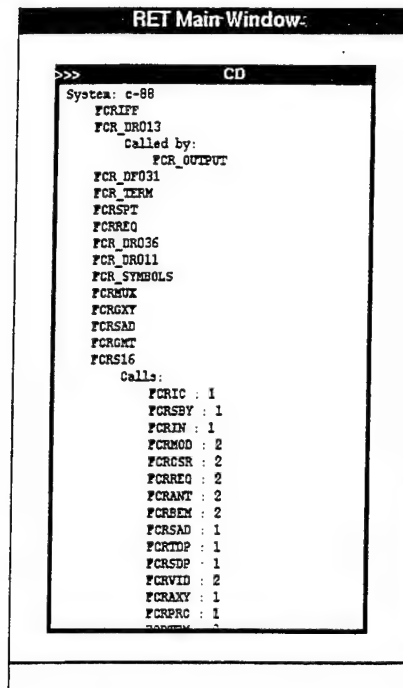
Figure 18 Expanded Call Diagram
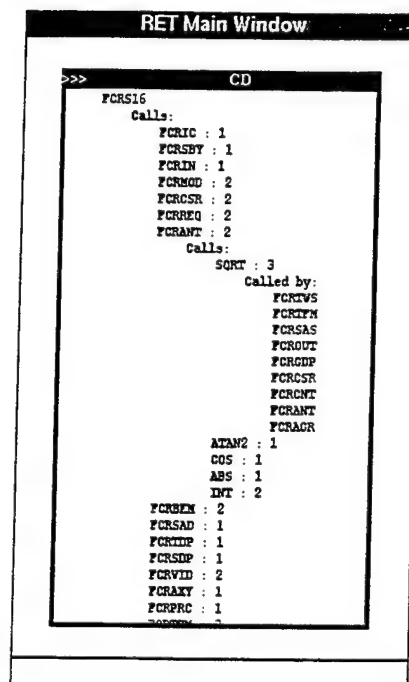
Figure 19 Further Expanded Call Diagram

46

Figure 19 shows the subprograms that FCRANT calls. It calls SQRT three times, for example. SQRT is called by FCRTWS and others. You may expand subprograms in the CD like this to any depth.

Table 12 lists the options in the menu that pops up when you right-click on a subprogram name in the CD view. **Hide Self** causes the subprogram to disappear from the display. The pop-up menu may show either the **Show Calls** or **Hide Calls** options depending on whether the CD already displays the subprograms that the selected subprogram calls. The pop-up menu may show either the **Show Called By** or **Hide Called By** options depending on whether the CD already displays the subprograms that call the selected subprogram. **View Source** pops up the FORTRAN Source Code Listing view for the selected subprogram. **PUP** and **Inspect** are debugging options. **Abort** closes the menu without taking any action.

Table 12 Call Diagram Pop-up Menu for Object

CD: OBJECT
Hide Self
Show Calls
Hide Calls
Show Called By
Hide Called By
View Source
PUP
Abort

Table 13 lists the options in the menu that pops up when you right-click on the background in the CD view. **Hide Self** causes the currently selected subprograms to disappear from the display. **Show Kids** causes the information nested below the selected subprograms to reappear if it has been hidden. **View Source** pops up the FORTRAN Source Code Listing view for the selected subprograms. **Deselect** causes all selected subprograms to become insensitive to the options that operate on selected subprograms, such as **Hide Self**. **PUP** and **Inspect** are debugging options. **Abort** closes the menu without taking any action.

47

Table 13 Call Diagram Pop-Up Menu for Background

CD: BACKGROUND
Hide Self
Show Kids
View Source
Deselect
PUP
Abort

Table 14 lists the options in the menu that pops up when you right-click on the window title bar of the CD view. The **Lower, Hide**, **Move**, **Reshape**, and **Refresh** options let you control the CD window. **Output** causes the RET prototype to prompt you for a file name and print the CD view to the file. **Hyperlink On** and **Hyperlink Off** are disabled in the RET prototype. **Hide Self** causes the selected subprograms to disappear. **Show Kids** causes the information nested below the selected subprograms to reappear. **Abort** closes the menu without taking any action.

Table 14 Call Diagram Pop-Up Menu for Window

CD: WINDOW
Lower
Hide
Move
Reshape
Refresh
Output
Hyperlink On
Hyperlink Off
Hide Self
Show Kids
Deselect
Abort

5.5   DECLARATION DIAGRAM

Figure 20 shows a sample Declaration Diagram (DED) as it appears immediately after generation. The "c-91" at the top indicates that the names of 91 subprograms and common blocks (called lines) follow. The "c-4 *" on the third line indicates that four lines are hidden below FCR. The "*" indicates that the lines under FCR are hidden, i.e., *not* shown. In contrast, the absence of an asterisk in "system c-91" implies that 91 lines *are* shown under "system."

A line without an asterisk is said to be expanded and a line with an asterisk is said to be contracted. The asterisk is a surrogate for the missing information. You may expand or contract a subprogram or common block by middle-clicking on it. Alternatively, you may right-click on it and choose the **Show Kids** or **Hide Self** pop-up menu options (see Table 15).

These and other options in the menus operate on the subprogram or common block that you right-clicked on to pop-up the menu, i.e., the selected line. You may also designate one or more selected lines by left-clicking on them. Left-clicking toggles the selected status of a line. The DED distinguishes selected lines by displaying them in reverse video.

Figure 20   Initial Declaration Diagram

Figure 21 shows the DED after expanding "ALL_SHRMEM common blocks" to show the sole common block (ALL_SHRMEM), and the single variable that it defines (SHRMEM). FORTRAN unfortunately allows multiple common blocks with the same name but different definitions. If ALL_SHRMEM had been multiply defined in this way, the DED would have shown more than one common block with that name, but it only shows one in this case.

Ten subprograms are selected in Figure 21. The first line reveals that a line is hidden somewhere in the system by the "h-1." Figure 22 shows the results of performing several operations on the DED in Figure 21. The "h-11" at the top of Figure 22 indicates that the 10 lines that were selected in Figure 21 are now hidden (in addition to the one line that was already hidden, for a total of 11). This change resulted from the **Hide Self** option of the pop-up menu in Table 16.

Figure 22 demonstrates the results of expanding the FCRDAT common block and two of the variables (INDUNZ and ISDFL) that it contains. The line "FCRDAT c-390" shows that the common block defines 390 variables. INDUNZ is an array of 10 integers, declared on line 266 in file AVM_FCR_LCM.FOR, and referenced on line 434 in file fcrtfm.for and in file fcrmux.for.

Figure 21   Declaration Diagram With Selected Subprograms
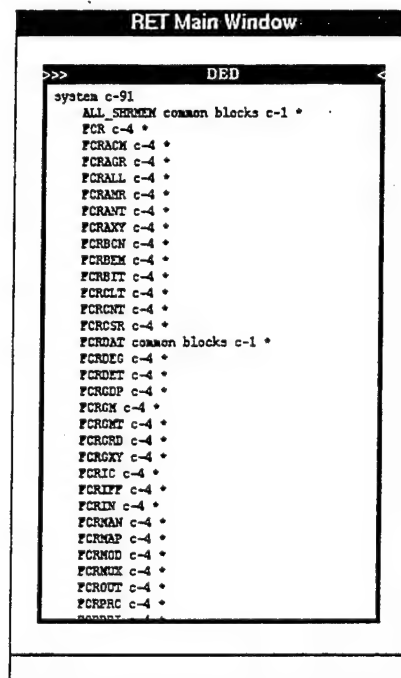
Figure 22 Declaration Diagram With Expanded Common Blocks

Figure 23 shows two expanded subprograms. Near the bottom and indented under FCRSPT are four lines that provide information on formal parameters, constants, variables, and include statements in the subprogram. FCRSPT has no formal parameters and declares no data objects, but it includes three files. Near the top, FCRSDP is shown to declare eight variables and include four files. The "variables" and "includes" lines are expanded to reveal the variable and include file names, respectively. Variable NJAM is also expanded.

Table 15 lists the options in the menu that pops up when you right-click on a line in the DED view. **Hide Self** causes the line to disappear from the display. **Show Kids** causes the information nested below the line to reappear if it was hidden. **View Source** pops up the FORTRAN Source Code Listing view for the line if it is a subprogram line. **PUP** is a debugging option. **Abort** closes the menu without taking any action.

51
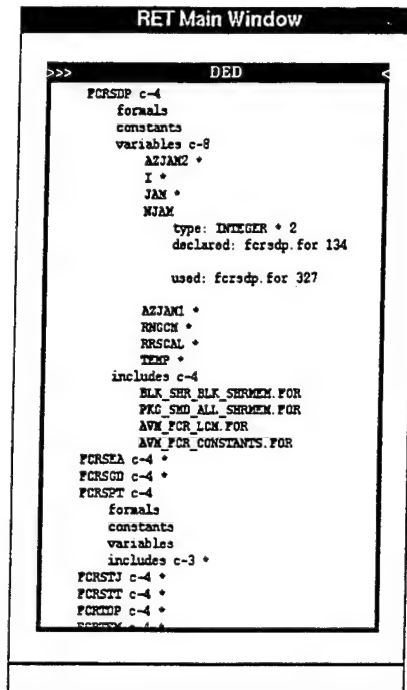
Figure 23   Declaration Diagram With Expanded Subprograms

Table 15 Declaration Diagram Pop-Up Menu for Object

DED: OBJECT
Hide Self
Show Kids
View Source
PUP
Abort

Table 16 lists the options in the menu that pops up when you right-click on the background in the DED view. **Hide Self** causes the currently selected line(s) to disappear from the display. **Show Kids** causes the information nested below the selected line(s) to reappear if it has been hidden. **View Source** pops up the FORTRAN Source Code Listing view for the selected line if it is a subprogram line. **Deselect** causes all selected lines to become insensitive to the options that operate on selected lines, such as **Hide Self**. **PUP** is a debugging option. **Abort** closes the menu without taking any action.

**Table 16 Declaration Diagram Pop-Up
Menu for Background**

DED: BACKGROUND
Hide Self
Show Kids
View Source
Deselect
PUP
Abort

Table 17 lists the options in the menu that pops up when you right-click on the window title bar of the DED view. The **Lower, Hide, Move, Reshape**, and **Refresh** options let you control the DED window. **Output** causes the RET prototype to prompt you for a file name and print the DED view to the file. **Hyperlink On** and **Hyperlink Off** are disabled in the RET prototype. **PUP** is a debugging option. **Abort** closes the menu without taking any action.

**Table 17 Declaration Diagram Pop-Up
Menu for Window**

DED: WINDOW
Lower
Hide
Move
Reshape
Refresh
Output
Hyperlink On
Hyperlink Off
PUP
Abort

5.6 FORTRAN SOURCE CODE LISTING

Figure 24 shows a sample FORTRAN, and the corresponding Ada Source Code Listing view. The *IF* statement near the center of the Ada view calls an external function (IS_SET) and an external procedure (RESET). The explicit type conversion of the return value of IS_SET to BOOLEAN is superfluous (see Figure 12). Section 5.7 explains how to edit the generated Ada source code to remove the type conversion.

Table 18 lists the options in the menu that pops up when you right-click on a FORTRAN statement in the FORTRAN Source Code Listing view. The **Select (Deselect) Statement** option makes the statement (in)sensitive to other options that operate on statements, such as the Translate options. The **Translate** options translate the selected statement(s) to Ada code and either insert the resulting Ada statements into the subprogram body on the RHS, print them to the *REFINE* Emacs buffer, or invoke the Inspector debugging utility on them, with respect to the order of the options given in the table. **Inspect**, **PUP**, **MCN**, and **PN** are debugging options. **Abort** closes the menu without taking any action.

Figure 24   Source Code Listings

54

**Table 18 FORTRAN Pop-Up Menu
for Statement**

FORTRAN SRC
Select Statement
Deselect Statement
Translate to Ada Body
Translate to Ada and Print
Translate to Ada and Inspect
Inspect
PUP
MCN
PN
Abort

Table 19 lists the options in the menu that pops up when you right-click on a FORTRAN subprogram in the FORTRAN Source Code Listing view. The **Select (Deselect) All Statements** option makes all statements in the view (in)sensitive to other options that operate on statements, such as the Translate options.

**Table 19 FORTRAN Pop-Up Menu
for Subprogram**

FORTRAN SRC
Select All Statements
Deselect All Statements
Show Parent Unit
Show Only This Unit
Show Corresponding Ada Body
Translate to Ada Body
Translate to Ada and Print
Translate to Ada and Inspect
Inspect
PUP
MCN
PN
Abort

55

Show Parent Unit causes the DED to show the compilation unit associated with the subprogram in lieu of the subprogram. The practical and visible effect of this option is to show the comments that precede the *SUBPROGRAM* statement. **Show Only This Unit** reverses the above operation, causing the DED to show the subprogram associated with the compilation unit in lieu of the compilation unit. The practical and visible effect of this option is to remove the comments that precede the *SUBPROGRAM* statement from the display. **Show Corresponding Ada Body** displays, in the Ada Source Code Listing view, the Ada body associated with the FORTRAN subprogram.

The **Translate** options translate the selected statement(s) to Ada code and either insert the resulting Ada statements into the subprogram body on the RHS, print them to the *REFINE* Emacs buffer, or invoke the Inspector debugging utility on them, with respect to the order of the options given in the table. **Inspect**, **PUP**, **MCN**, and **PN** are debugging options. **Abort** closes the menu without taking any action.
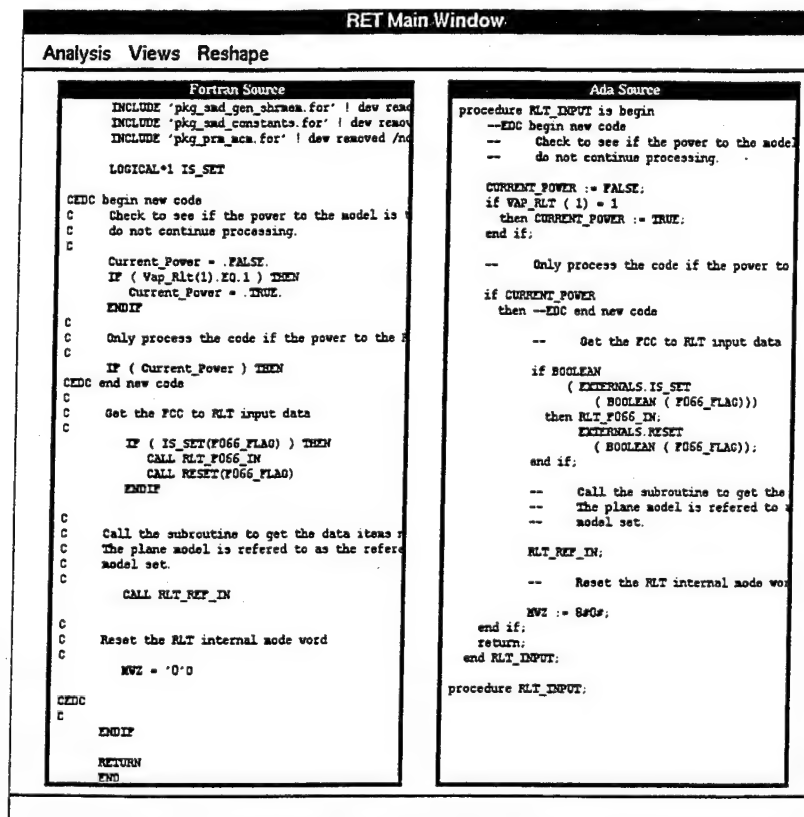
Table 20 lists the options in the menu that pops up when you right-click on the background in the FORTRAN Source Code Listing view. **Deselect All** makes all statements in the view insensitive to other options that operate on statements, such as the Translate options in Table 19. **Inspect**, **PUP**, **MCN**, and **PN** are debugging options. **Abort** closes the menu without taking any action.

**Table 20 FORTRAN Pop-Up Menu
for Background**

FORTRAN SRC BACKGROUND
Deselect All
Inspect
PUP
MCN
PN
Abort

Table 21 lists the options in the menu that pops up when you right-click on the title bar of the FORTRAN Source Code Listing view. The **Lower, Hide, Move, Reshape, Refresh,** and **Redraw Window** options let you control the window. **Output Window** causes the RET prototype to prompt you for a file name and print the FORTRAN source code to the file. **Hyperlink On** and **Hyperlink Off** are disabled in the RET prototype. **Inspect, PUP, MCN,** and **PN** are debugging options. **Abort** closes the menu without taking any action.

**Table 21 FORTRAN Pop-Up Menu
for Window**

FORTRAN SRC WINDOW
Lower Window
Hide Window
Move Window
Reshape Window
Refresh Window
Redraw Window
Output Window
Hyperlink On
Hyperlink Off
Inspect
PUP
MCN
PN
Abort

5.7   ADA SOURCE CODE LISTING

Figure 25 shows the FORTRAN Source Code Listing view from Figure 24, and a modification of the Ada Source Code Listing view from Figure 24. You may modify the Ada source code by choosing the **Edit Body** option from Table 3, Table 4, Table 8, or Table 9, editing the resulting Emacs buffer, saving the file, and then choosing the **Analyze Ada Files** option from Table 7 or Table 11.

We reformatted the Ada source code in Figure 25 via Emacs, but you may modify the source code file on the hard disk by any means, including the execution of an Ada pretty printer (if one is available; the RET prototype doesn't provide one) outside of the RET prototype. We rearranged the comments and indentation, and removed the superfluous BOOLEAN type conversion.

Table 22 lists the options in the menu that pops up when you right-click on an Ada statement in the Ada Source Code Listing view. The **Select (Deselect) Statement** option makes the statement (in)sensitive to other options that operate on statements, such as the **Delete Statements** option. The **Move Statements Before** and **After** options let you

57

Figure 25   Modified Ada Source Code Listing

reorder the statements by moving the selected statements before or after the statement. **Delete Statements** removes the selected statements from the display. (If you modify an Ada Source Code Listing view via the **Delete** or **Move** options in Table 22, you must choose the **Redraw** option in Table 24 to update the view or you will not see the changes.) **Inspect, PUP, MCN**, and **PN** are debugging options. **Abort** closes the menu without taking any action.

Table 22 Modified Ada Source Code Listing

ADA SRC
Select Statement
Deselect Statement
Move Statements Before
Move Statements After
Delete Statements
Inspect
PUP
MCN
PN
Abort

Table 23 lists the options in the menu that pops up when you right-click on an Ada subprogram body or specification in the Ada Source Code Listing view. The **Select (Deselect) All Statements** option makes all statements in the view (in)sensitive to other options that operate on statements. **Show Parent Unit** causes the view to show the compilation unit associated with the subprogram body or specification. The practical and visible effect of this option is to show the Ada package that declares the subprogram body or specification. **Show Only This Unit** reverses the above operation, causing the view to show the subprogram body and specification associated with the compilation unit. The practical and visible effect of this option is to focus in on the declarations of a specific subprogram in the package. **Inspect, PUP, MCN,** and **PN** are debugging options. **Abort** closes the menu without taking any action.

**Table 23 Ada Pop-Up Menu
for Subprogram**

ADA SRC
Select All Statements
Deselect All Statements
Show Parent Unit
Show Only This Unit
Inspect
PUP
MCN
PN
Abort

Table 24 lists the options in the menu that pops up when you right-click on the background in the Ada Source Code Listing view. **Deselect All** makes all statements in the view insensitive to other options that operate on statements, such as the **Delete Selected Statements** option. **Delete Selected Statements** removes the selected statements from the view. **Inspect, PUP, MCN,** and **PN** are debugging options. **Abort** closes the menu without taking any action.

**Table 24 Ada Pop-Up Menu
for Background**

ADA SRC BACKGROUND
Deselect All
Deselect Selected Statements
Inspect
PUP
MCN
PN
Abort

Table 25 lists the options in the menu that pops up when you right-click on the title bar of the Ada Source Code Listing view. The **Lower, Hide, Move, Reshape, Refresh,** and **Redraw Window** options let you control the window. (If you modify an Ada Source Code Listing view via the **Delete** or **Move** options in Table 22, or the **Delete Selected Statements** option in Table 24, you must choose the **Redraw** option in Table 25 to update the view or you will not see the changes.) **Output Window** causes the RET prototype to prompt you for a file name and print the Ada source code to the file. **Hyperlink On** and **Hyperlink Off** are disabled in the RET prototype. **Inspect, PUP, MCN,** and **PN** are debugging options. **Abort** closes the menu without taking any action.

Table 25 Ada Pop-Up Menu for Window

ADA SRC WINDOW
Lower Window
Hide Window
Move Window
Reshape Window
Refresh Window
Redraw Window
Output Window
Hyperlink On
Hyperlink Off
Inspect
PUP
MCN
PN
Abort

REFERENCES

1. D.E. Wilkening, Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume I, Project Summary, Account, and Results, TASC Technical Report TR-6661-4, TASC, Inc., Reading, Massachusetts, 17 April 1995.

2. D.E. Wilkening, Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume II, Reenginering Tool (RET) Diagrams, TASC Technical Report TR-6661-5, TASC, Inc., Reading, Massachusetts, 17 April 1995.

3. plusFORT Reference Manual, Revision B, Polyhedron Software Ltd., Standlake, Witney, UK, 1993.

4. VAX FORTRAN Language Reference Manual, Digital Equipment Corporation, Maynard, Massachusetts, Order Number: AA-D034E-TE.

5. Aho, A.V., Sethi, R., and Ullman, J.D., Compilers — Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, 1988.

6. Chikofsky, E.J., and Cross II, J.H., Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, p. 13-17, January 1990.

7. Cross II, J.H., Chikofsky, E.J., and May Jr., C.H., Reverse Engineering, *Advances in Computers 35*, p. 199-254, 1992.

8. Byrne, E.J., Gustafson, D.A., A Formal Process Model for Software Reengineering: The Analysis Phase, Kansas State University Technical Report TR-CS-91-12, 12 November 1991.

9. Hutchens, D. and Basili, V.R., System Structure Analysis: Clustering With Data Bindings, IEEE Transactions on Software Engineering, SE-11(8), p. 749-757, August 1985.

10. Muller, H.A., Orgun, M.A., Tilley, S.R., and Uhl, J.S., A Reverse Engineering Approach to Subsystem Structure Identification, *Journal of Software Maintenance: Research and Practice*, 5(4), p. 181-204, December 1993.

11. GNU Emacs Manual, Seventh Edition, Version 18, Free Software Foundation, September 1992.

12. REFINE/FORTRAN User's Guide, Reasoning Systems, Inc., Palo Alto, CA, 5 November 1992.

13. INTERVISTA User's Guide, Reasoning Systems, Inc., Palo Alto, CA, 4 March 1991.

APPENDIX A
ACRONYMS FOR VOLUME I

ASG – Abstract Syntax Graph

ASRET – Avionics Software Reengineering Technology

ASTS – Avionics Software Technology Support

CD – Call Diagram

CS – Common Clients and Suppliers Metric

DED – Declaration Diagram

DFD – Data Flow Diagram

ECS – Embedded Computer System

IS – Interconnection Strength Metric

LHS – Left-Hand Side

LRM – Language Reference Manual

PACK – Packager View

RET – Reengineering Tool

RHS – Right-Hand Side

SCL – Source Code Listing

WL/AAAF-3 – Software Concepts Group, Avionics Logistics Branch, Wright Laboratory